HI-RES SECRETS

INCLUDING
BLOCK SHAPES FOR
APPLESOFT* OR ASSEMBLY

Also Vector Shapes, Hplot-Shapes, Text File Shapes; ANIMATION of many types, including Logical Shift, XDRAW, Page-Flipping; Instant Graphics and Shape-Drawing Programs; Machine Language Animation for all types of shapes; Font Programs; Machine Language Sounds; Hi-res Scrolling; Color Animation; Music (Write:Record:Play); YTABLE; Assembling; Utilities; *HI-RES COLOR PALETTE, PAINTBRUSH & COLOR-FILL PROGRAMS*

**BY
DON FUDGE**

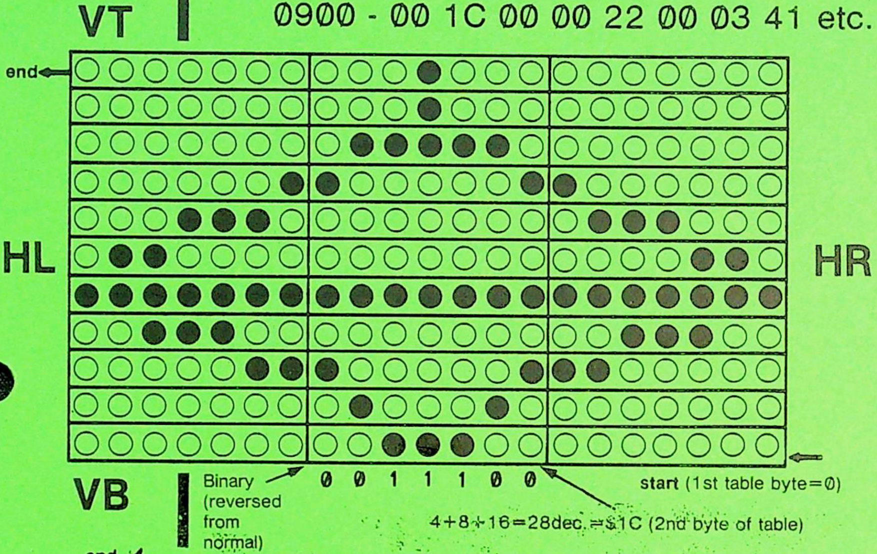
"IF YOU CAN'T BUDGE IT, FUDGE IT"

**4
Disks**
Applesoft 48K

BLOCK-SHAPE

Data array (3X11) = 33 bytes
Binary File: (in Hex.)

Ø9ØØ - ØØ 1C ØØ ØØ 22 ØØ Ø3 41 etc.



Binary (reversed from normal)

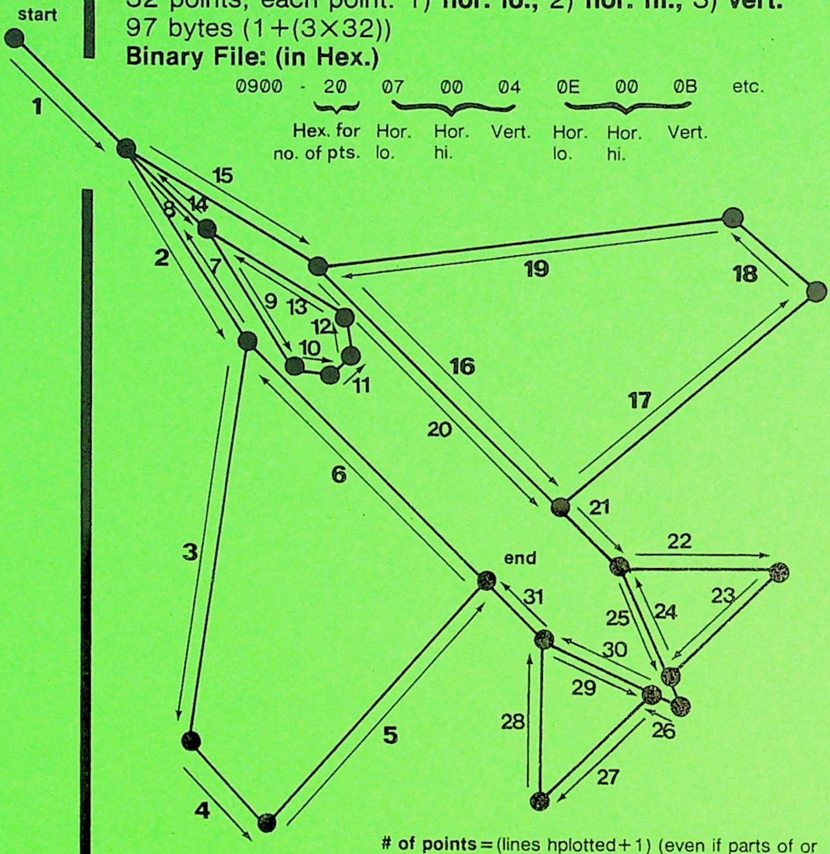
start (1st table byte=Ø)

4+8+16=28dec.=$1C (2nd byte of table)

HPLOT-SHAPE

32 points; each point: 1) **hor. lo.,** 2) **hor. hi.,** 3) **vert.**
97 bytes (1+(3×32))
Binary File: (in Hex.)

```
0900 -  20        07    00    04      0E    00    0B    etc.
       Hex. for   Hor.  Hor.  Vert.   Hor.  Hor.  Vert.
       no. of pts. lo.  hi.           lo.   hi.
```



# of points = (lines hplotted + 1) (even if parts of or all of some of the lines get drawn twice or more)

VECTOR SHAPE

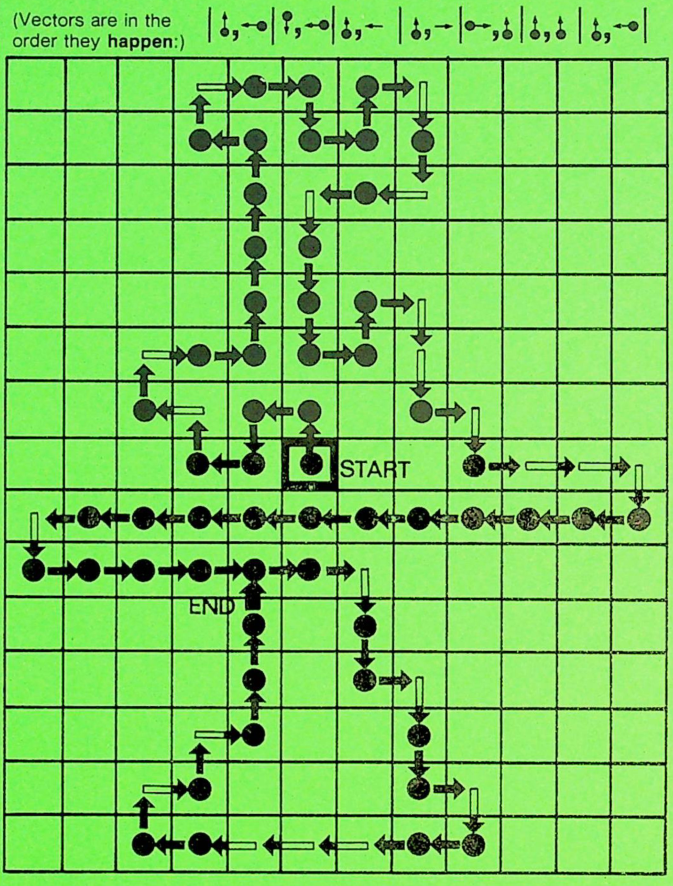
36 vector bytes

"THE HUNTER"

Binary File: (in Hex.) (without index)

```
9Ø21-   3C   3E   1C   ØC   25   24   3C   etc.
```

(Vectors are in the order they happen:)



HOW TO TRY OUT HI-RES SECRETS:

(A) There are 2Ø9 programs/files on the 4 disks that come with this system, and there are hundreds of pages of information in this manual. It would take a month to truly "try out" HI-RES SECRETS. But let's attempt to guide you in trying out this system (a small part of it) as best we can in the space of a few pages of instructions. Use the experience to familiarize yourself with this package and give yourself ideas about how you'd like to begin utilizing it. Look through the table of contents now and also the Index(?) to see some of the areas covered.

(B) Virtually all machine language programs/routines in this system are completely explained, step by step, in a clear fashion that assumes only that you know what Applesoft Basic is about, so if any hi-res area is found to be especially intriguing at this time (such as the popular Color Fill-In algorithms), then turn to that section of the manual and boot the disk that deals with that subject (28D) and test, learn, study, draw, color, paint, design, etc., to your heart's content.

(C) Now for the general try-out:

(**1**) Boot 28A, check our introduction if you haven't seen it.

(**2**) Once you get to the menu, choose #8.

(**3**) Give QQ as shape table name, and keep watching the upper left corner of the screen.

(**4**) You've just seen a shape being drawn from a simple text file. Now let's scan a regular vector shape and put it into a text file.

(**5**) Go to menu by saying you want no more shapes. Choose 7. Choose vector shape table called MAN and shape #5. You'll **scan** this shape and save it in a text file.

(**6**) Give a height of 18 and a width of 6 (bytes). You'll see the block of memory that defines your shape now. It'll be outlined in white. Answer yes, you want to save it, and give QB as the shape name.

(**7**) Go to menu and choose #8. Give QB as shape table name. Keep your eyes on the upper left of the screen. **Scanning** in reverse is drawing. Most block-shapes are stored in binary files and BLOADed for use. Try #9 in menu with QB --- your block-shape is upside down now. Block-shapes are always rectangular blocks of memory.

(**8**) Now let's make a binary block-shape from a vector shape (Vector Shapes are explained in your Applesoft manual on pages 91-1ØØ): Choose #4 on the menu. Hit RETURN to erase screen, once in program, and then choose option #1.

(**9**) Give MAN as the shape table name and 32768 as the table address. Ask for shape #1 with X coordinate (horizontal coordinate) of 2Ø and Y coordinate (vertical) of 2Ø. Say **no** more shapes.

(**10**) Hit option 7 from menu of this 4 of 28A program. This will allow you to paddle-define a block-shape for creation of a binary file shape table. Move paddles until you have X=11 and Y=7. Hit paddle zero button until it beeps. Move paddles until dot is at X=29 and Y=30. Hit paddle button #1. Your block shape is 3 by 23. When asked if the rectangle is done okay, say Y for yes. (If the shape isn't really clear, turn the color off on the screen so that only black and white show.) There will be other questions, and the shape isn't yet saved, but there's no room on this disk for it, so either put in an initialized blank disk with the same DOS or reboot disk.

(11) Boot 28C.

(12) Go to #5. We'll convert a block-shape to a vector shape --- the latter is easiest to scale up or down, rotate, or draw in many colors. You'll notice that hplot-shapes may also be converted into vector shapes. Hit RETURN to erase. Hit N for no HPLOT-SHAPE.

(13) Answer no to paddle-defining what's on screen. (If you hadn't erased the screen on entering program, you could turn the walking man into a vector shape.). Ask for shape table Q3, shape #1. Give VT=2, VB=2Ø, HR=7, HL=2. Remember the block-shape rectangle?

VT
HL □ HR
VB

(Don't use HL of Ø in this program)
(Don't switch disks.)

Well, V means vertical, T means top, H means horizontal, B means bottom, L means left, R means right. With these numbers, you can put a block-shape anywhere on the screen. Make sure VB-VT=height and HR-HL=width in bytes (not dots). The flying saucer is 18 high and 5 wide.

(14) Move the paddles until X=12 and Y=2 and hit PDL button #Ø, then move dot to X=42 and Y=2Ø and hit PDL button #1. Say the rectangle's okay, once you see it. The scanning you see turns screen bytes into plotting vector bytes.

(15) Now hit RESET and type HCOLOR=3:HGR:SCALE=1: ROT=16:DRAW 1 AT 2ØØ,1ØØ. Hit RETURN. If you needed a block-shape to be at a different rotation, you could do what you just did and either use that shape as a vector shape from now on or go to 4 of 28A and save it (without erasing screen!) as a block-shape again.

(16) Reboot 28C and run #4. Choose delay loop of 1. RESET to quit. 114 bytes load in per shape-draw. The shape moves only 1 dot sideways per move. The hi-res screens are flipped back and forth 87 times a second (between page one,(HGR,) and page two, (HGR2). This means that 22272 bytes get loaded from a block-shape table to screen addresses every second. This is at least 20% faster than utilization of HPOSN (an Applesoft routine). This program uses YTABLE look-up rather than HPOSN. If this is Greek to you, don't worry --- it's all explained in the manual. The shape would flicker if we didn't flip back and forth between hi-res screens 1 & 2. Most animation has this flipping. You'll see how it's done in this manual. You'll really be able to understand the whole animation process **clearly** --- which few or no other information sources facilitate.

(17) Now run 2 of 28C. It's a sample game that uses NOISES, VIOLIN, EXPLOSION (CALL 5472) and lots of other sounds, as well as simple one-page vector animation for explosions. You may list it out or change the program or just play it and have fun. You'll be able to use all kinds of sounds, noises, tones and songs in **your** programs very soon. They're simple to do. This system contains dozens of ready-to-use sounds and shape tables and machine language animation routines and shape sequence creators (automatic) and automatic fill routines and paint routines to color pictures or shapes with.

(18) Unless you want to play with the music program on this disk, boot 28B now and choose B which is **Superfont.** Read instructions and choose various styles, scales, and colors of font characters. You may save your creations on a printer or merely on disk --- or both. Notice the fast 64-line scrolling --- that's a machine language routine being CALLed. Other scrolling routines for scrolling the screen in other directions are also included and explained.

(19) Run 3 of 28B. Choose R for RIGHT and choose the shape table SQSEQC, and use shape #1 and VT of 0, VB of 32, HR of 6, and HL of 0. If you have a color monitor, you'll notice that the color stays the same even though we're only moving 1 bit at a time. What's happening here is the shape is being shifted twice on one page of hi-res white the other page is displayed --- and then we flip pages and do the opposite. The type of shifting in this animation is called a rotation, which is merely a logical shift of all bits in a byte either left (ROL) or right (ROR). You needn't make block-shape shape sequences if you do shift-animation, although it's slower. Incidentally, the smaller the shape the faster the animation can be.

(20) Now boot 28D and run #4 in the menu --- this is my Palette program. It's really fun to play with, especially if used in conjunction with 2 of 28A or 8 of 28D. Hit RETURN to erase screen.

(21) Hit 12 (and RETURN, obviously) for full-screen and 9 for shape loading, and option 3 for picture and leave the disk in. The picture name is COMPOSITE 2. Hit 1 to see screen, any key for menu. The picture was drawn with 2 of 28A mostiy --- it **all could** have been done with 2 of 28A easily. I'm used to using our **Super Shape Draw and Animate** package for shape drawing, so that's what I drew the man with.

(22) Now hit 10 and adjust your TV color so that the first **large** square drawn is green. Move the dot so that the third to the last bottom square is centered on --- orange. Hit PDL button #0 and say yes. (The palette draws only once --- after that you find it instantly by choosing option 10. The disk is much too full to have allowed me to load in the palette as a picture from a 34-sector binary file picture.)

(23) Type 16 and return. The color byte for even addresses (for orange) is 170. For odd addresses it's the binary complement (not counting hi bit) of 170 --- 213. We'll be using 4 adjacent color bytes stored in $6-$9 of memory. The bottom 2 bytes are often different from the top ones --- this makes many more colors possible.

(24) Choose option 11 in the menu to fill. Read the instructions and continue. Put the dot in the very center of the doorway and press the #0 button. The program wastes little time! Fill a few other squares and then hit button #1 to exit.

(25) Choose option 17. You now have a mystery color/pattern. Choose 11 and fill a couple squares with it. Then hit button #1 and choose option 9 and then choose option 2 in that section. Choose shape table: POS. Choose shape #2 and put it up in the upper left corner using button #0, hit any key, then hit Y for another.

(26) "Draw" him in the top half of the door. No good! Now try saying no more shapes and changing the color using option 5 of the menu. Choose color #4. Now choose option 9 and use option 2 of 9 and use table POS again --- choose shape #3. Center dot on bottom half of door and hit button #0. See how shapes look better if you don't mix **hi-bit on** and **hi-bit off** colors? Hcolor 4 is black and its byte is 128. The color bit (#7) is ON in colors 4-7 and off in colors 0-3. Colors whose bytes are 128 decimal ($80 hex) or more have their color bit **set**, or ON, meaning it's a one, not a 0. The decimal value of the 1 in the 8th bit (#7, since the first bit is numbered 0) is 128. The reason colors with opposite color-bit settings don't mix is that for every byte, the high bit **must** be either on **or** off.

(27) Choose 10 again and pick a color and choose 11 and fill a few more squares or sections. Really odd shaped ones may require more than one filling. Now hit SPACE BAR, then 0 for paintbrush color. Now hold down button #1 and move the dot until it's at the left edge of the door. Slowly move PDL #0 clockwise. You may erase some or all of the doorway color, the shapes in the doorway, etc. You may build a window and/or re-color that doorway. Use a delicate touch --- use skill. Hit SPACE BAR, P, RETURN, 6, and 9. You'll find that blue, another **hi-bit-on** color, looks nice with orange. It's the complement of orange, which means trade all ones for 0's and 0's for ones (except the high bit).

(28) Hit SPACE BAR and F to fill. Hit A to abort clicks. Fill the "window" you just made. Hit PDL button #1.

(29) Hit 0 to go to main menu. Choose 8. Hit **SPACE BAR** to enter program. Choose #1 to see screen --- yep! Still there. Choose 9 for shapes and then 2 for vector shapes and then choose shape table name of ANIMALS, and use shape #6. Move shape, with paddles, to a clear space on wall and hit button #1 and then move away. It got copied! Hit PDL button 0. Choose no more shapes and choose 14 to complement every byte on the screen.

(30) Pretty wild, right? Now choose 16 and set all color bits. Then choose 13 and filter out various colors (best to filter orange or blue when hi bits set and violet or green when hi bit off).

(31) Now go to 0 of menu and run 9 of 28D. This program does an incorrect automatic fill on an hplot-shape. Hit RESET and list out the program. 950 is the workhorse, 1000 is the FOR-NEXT showing **how many** "fills" to do, 2000 has the data which 1010 reads in. The data is simply 2 coordinates (X, Y) and 4 color bytes per fill. The incorrectness, used for demonstration, is wrong fill locations, wrong number of fill locations, and bad mixture of "hi" and "lo" colors. (For more details on 9 of 28D see first part of Chapter 29.)

(32) Now boot 28D and run A. This is how the use of the fill routine to create "instant scenes" is **supposed** to look. List out A of 28D. Nothing very complex here. About 13 fillings per second are happening; let this program be your example to follow when you create your own "instant scenes".

HI-RES SECRETS

*** by Don Fudge ***

"If you can't Budge it, Fudge it."

To use with Apple* II (or III) Computers
with 48K and Applesoft* in ROM
(set includes disks 28A, 28B, 28C, 28D)

(Includes Instant Graphics (Block-Shapes) drawing card and also the music codes card for Music (Write:Record:Play))

*Apple and Applesoft are registered trademarks of Apple Computer, Inc.

AVANT-GARDE CREATIONS
P.O. Box 30160
Eugene, OR 97403
(503) 345-3043 (12 noon to 6 PM, 7 days a week)

First Printing September 1981
Second Printing November 1981
ISBN: 0-930182-21-9 (Book)
0-930182-22-7 (Package)

Printed in the United States of America

REQUEST FOR ACKNOWLEDGEMENT . . .

Avant-Garde Creations respectfully requests that those persons using routines from HI-RES SECRETS in their own programs include acknowledgements of the source of these routines (i.e. "The color-fill and animation routines are from HI-RES SECRETS by Avant-Garde Creations.")

No written permission or royalty is requested. However, we would very much like to know how people are implementing the routines in their own programs, with the added possibility that we would advertise your game or program as one that incorporates our routines.

We thank you in advance for this courtesy.

DISCLAIMER OF WARRANTIES

No warranty, either expressed or implied, is made with respect to the enclosed computer software package or any part thereof, its quality, performance, merchantability, or fitness for any particular purpose.

ACKNOWLEDGEMENT

Avant-Garde Creations would like to thank the following people for their willingness to communicate in magazines, in writing, or by phone. Willingness to share information about Apple* Computer programming is necessary and important for the future viability of the Apple* II microcomputer. Thanks to: Bob Bishop, C.K. Mesztenyi, R.H. Good, Loy Spurlock, Stephen R. Bergren, Steven Dompier, Val J. Golding, Don Williams, R.M. Mottola, Ray McVay, Pat Connelly, Ted Perry, and others.

*Apple is a registered trademark of Apple Computer, Inc.

This book is dedicated to everyone who wishes to, like Avant-Garde Creations, help the world work better via the creative application of computers.

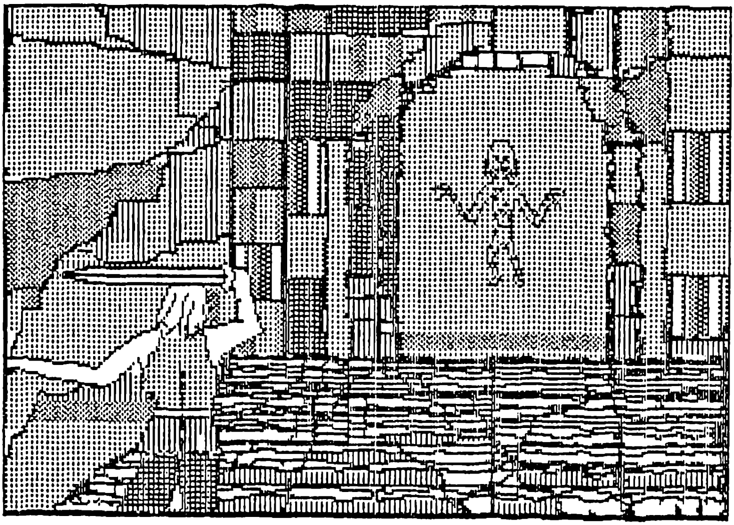


TABLE OF CONTENTS

INTRODUCTION

The purpose of this book is to help bridge the gap between novice programmers and expert programmers, especially in the area of high-resolution shapes, animation, sounds, fonts, assembly language, colors and color fill-in.

The people who know the most about assembly hi-res shape and animation routines are the people who are communicating the least (about what they know). These people are, understandably, writing fast-action arcade-type games and making lots of valuable green paper rectangles, also known as money. They have no intention of sharing their knowledge in a way where others would soon learn how to make games like they do. And with all the piracy and cheating going on in the software industry, who can blame them for being a bit cautious?

However, in spite of all the above it really is time that someone tried to put together a book about shapes, animation sounds, etc.

For people who want to get into Apple DOS there's *Beneath Apple DOS,* by Don Worth and Pieter Lechner (Quality Software), and for people who want to learn machine language programming but now know only BASIC there's *Apple Machine Language* by Don and Kurt Inman (Reston Publishing Company). For people who want to get deeper into 6502 assembly there's *6502 Assembly Language Programming* by Lance A. Leventhal (Osborne/McGraw-Hill, Inc.) or *Programming The 6502* by Rodney Zaks (Sybex). Also recommended: *Apple Orchard, Call Apple, Nibble, Micro,* and *Creative Computing.* Finally, I recommend a LISA Assembler (Programma).

The reason I say it's time someone wrote a decent shape (etc.) manual is that there's a tremendous duplication of effort going on lately and there's a lot of confusion about Hi-res, Assembly, and combining these two. Thousands of people are all concomitantly trying to figure out Apple graphics, and for some, the deeper into it they dig, the more confused they become. Once you're past the simple-shape-drawing stage, there's no longer any real source of information to support your further development as an Apple graphics programmer. Well...there ought to be.

Why should everyone have to start from scratch about all this? Why shouldn't there be decent reference materials handy, and understandable, fully-explained shape drawing and animating routines as well?

Computers are the most wonderful thing I've ever bumped into, and the most fun, and I learned a long time ago that good fun becomes even better when it's shared. Let's share this fun. You only live once...let's have a real ball with all this....

Perhaps Sir Isaac Newton isn't the only one who gets inspired upon being bonked by an Apple....

WHO IS DON FUDGE? 2

No, Don Fudge had nothing to do with *Raster Blaster* --- that was Bill Budge. Neither you nor I can make a *Raster Blaster,* so it follows that my only advice can be: IF YOU CAN'T BUDGE IT, FUDGE IT.

And yes, I am a sweet fellow.

But now let's look at programming intentions. I've little desire to program *like* other people do, or make a 999th version of *Space Invaders* and the like. At the end of my programming career I'd like to turn around, smile at the audience through my long grey beard, and sing "I Did It My Way."

A computer used in its best way is an extention of both one's rational and one's creative mind/ability/potential. It should allow you to do all those wonderful things you've always wanted to do but never had a way to do before. It should turn your fantasies into realities. It should manifest the truth of the following: "If you can imagine it, you can do it."

A computer can be a type of heaven, a Utopian friendship, and a creative adventure all rolled up into one. It can be a way of expressing parts of yourself you never even knew you had.

A computer is like finding a naked lady in your closet --- you don't know what it's all about but you're very open to learning.

So who is Don Fudge? Don Fudge is me. Don Fudge is you. Don Fudge is our imaginary playmate, our cuddly little micro-mirage, our rambunctious little RAM-chip rebel, our computerized guardian angel.

When the universe quits expanding, and the apex of universal movement is reached, and it commences what some would call its inevitable contractions (universal labor) and all matter gets closer and closer until finally it becomes an irresistible ONENESS, then the primeval Egg will explode again, and for billions of years our universe will expand again --- and yet through all this finite infinity will live in spirit the essence of one irrepressible entity, an entity which will permeate all spheres of reality and all manners of being, and will often cause the paternal representatives of developing intelligent life to remark with solemnity, and with an ancient twinkle in their benevolent eyes, to their computer-loving female offspring: "Yes, Virginia, there is a Don Fudge."

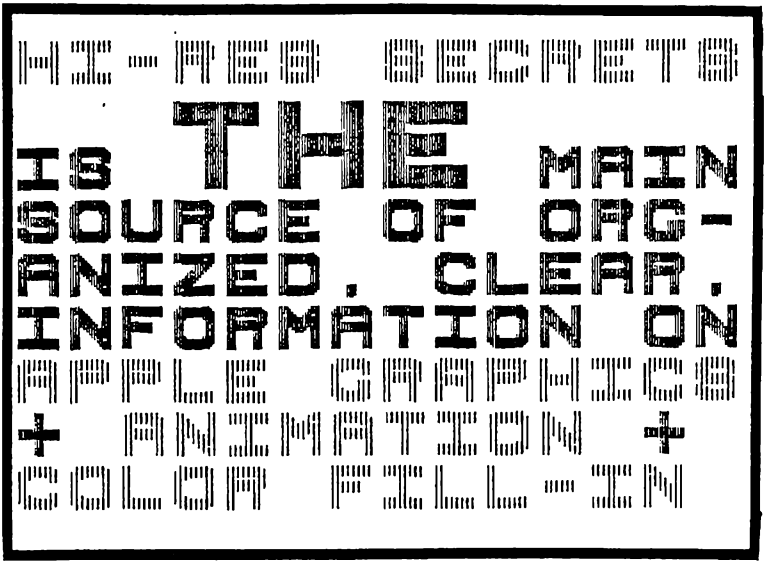
3 HOW TO USE THIS BOOK IF YOU DIDN'T BUY THE DISKS THAT COME WITH IT

We don't plan to sell the manual without disks as a regular thing, but if someone is willing to pay enough we'd probably part with a manual without disks. This isn't recommended though. You'll get a lot of useful information and a lot of source code list-outs and a few program list-outs, but the 4 disks that come with the manual are crammed with humongous amounts of programs, machine language files, shape tables, shape table creators, shape table sequence creators, source codes, color graphics, utilities, drawing programs, sample games, sound effects, music writing and violin programs, YTABLE programs...and you'll get listings for only a relatively small portion of all this if you merely get the book without disks.

The best reason to buy the book without disks is you have either no computer or a different brand of computer. The general ideas of what I'm doing and how should still come through to you.

DIRECTORY OF PROGRAM LISTINGS IN THIS BOOK

(Source codes, Applesoft programs, binary files, etc.)



HI-RES SECRETS

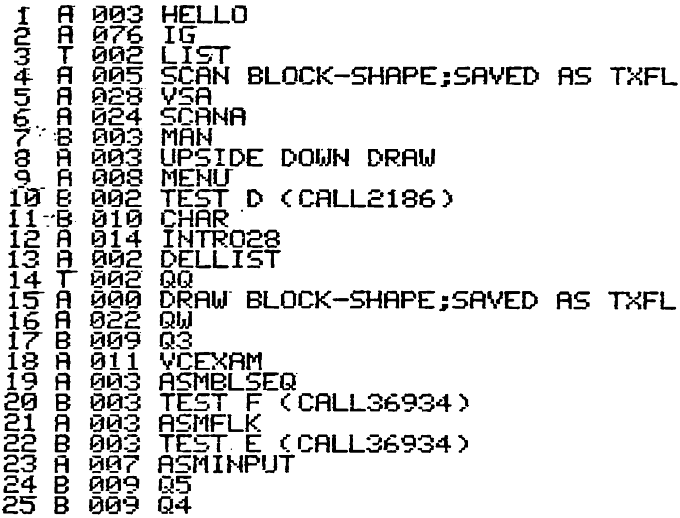
IS THE MAIN
SOURCE OF ORGANIZED. CLEAR.
INFORMATION ON
APPLE GRAPHICS
+ ANIMATION +
COLOR FILL-IN

5 DIRECTORY OF PROGRAMS ON THESE DISKS

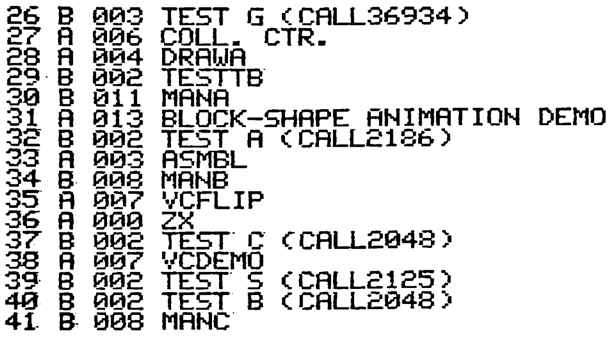
First, we'll look at the menu for 28A:

1) edit block shapes
2) draw shapes (solid or outline) and save/retrieve your creations
3) view shapes or block-shape animation
4) create block-shapes starting with anything, create sequences
5) view block-shapes
6) block-shape animation demo
7) scan vector shape, save as text file block-shape
8) view text file block-shapes
9) view text file block-shapes upside down
A) create block-shape sequences automatically from one vector shape
B) 2-page block-shape demo
C) 1-page block-shape demo
D) 2-page block-shape demo --- with user inputs
E) collision counter demo
F) fast block-shape demo
G) 1-page vector shape demo
H) 2-page vector shape demo
I) examine vector shapes

Next we'll look at the files on 28A:



```
 1 A 003 HELLO
 2 A 076 IG
 3 T 002 LIST
 4 A 005 SCAN BLOCK-SHAPE;SAVED AS TXFL
 5 A 028 VSA
 6 A 024 SCANA
 7 B 003 MAN
 8 A 003 UPSIDE DOWN DRAW
 9 A 008 MENU
10 B 002 TEST D (CALL2186)
11 B 010 CHAR
12 A 014 INTRO28
13 A 002 DELLIST
14 T 002 QQ
15 A 000 DRAW BLOCK-SHAPE;SAVED AS TXFL
16 A 022 QW
17 B 009 Q3
18 A 011 VCEXAM
19 A 003 ASMBLSEQ
20 B 003 TEST F (CALL36934)
21 A 003 ASMFLK
22 B 003 TEST E (CALL36934)
23 A 007 ASMINPUT
24 B 009 Q5
25 B 009 Q4
```



```
26 B ØØ3 TEST G (CALL36934)
27 A ØØ6 COLL. CTR.
28 A ØØ4 DRAWA
29 B ØØ2 TESTTB
30 B Ø11 MANA
31 A Ø13 BLOCK-SHAPE ANIMATION DEMO
32 B ØØ2 TEST A (CALL2186)
33 A ØØ3 ASMBL
34 B ØØ8 MANB
35 A ØØ7 VCFLIP
36 A ØØØ ZX
37 B ØØ2 TEST C (CALL2Ø48)
38 A ØØ7 VCDEMO
39 B ØØ2 TEST S (CALL2125)
4Ø B ØØ2 TEST B (CALL2Ø48)
41 B ØØ8 MANC
```

1) Hello

2) Instant Graphics (Block Shapes)

3) text file used in listing out protected programs

4) 7 above

5) 3 above

6) 4 above

7) MAN,A2352,L33Ø,8 V(vector) shapes (#5 is flying saucer), has seq., use in 7,A

8) 9 above

9) Menu

10) A2Ø48,L21Ø, use in C

11) CHAR,A$8DØØ,L$8DØ(A36Ø96,L2256),52 V shapes, use in intro.

12) introduction, uses CHAR

13) use in listing out protected programs

14) QQ 1 textfile shape,25HX4W,1Ø2 data in 2-dim. array

15) 8 above

16) A above

17) Q3,A23Ø4,L1792 (saucer),7B(block)shapes,184X5W,step 1,seq.,use in C,B,D or 28A or 4 of 28C

18) I above

19) C above

20) A36864,L324,use in D

21) B above

22) A36864,L288, use in B

23) D above

24) Q5,A23Ø4,L1792,("LL"),7B shapes, 41X8 but say 21HX8W,step2,seq.,use in D

25) Q4,A23Ø4,L1792,(submarine),7B shapes,21HX8W,step 1,seq.,use in D

26) A36864,L342,use in D

27) E above

28) 5 above

29) A2Ø48,L14Ø,use in 1,3,4,5,6,and A

30) MANA,A23Ø4,L237Ø,1Ø B shapes,21HX2W,(#5 is 15X3),step 1,seq.,use in 3,6,D,F

31) 6 above

32) A2Ø48,L189,use in F

33) F above

34) MANB,A23Ø4,L1624,delay:7Øhi,255lo,7B shapes,21HX3W,step 1,seq.,use in D

35) H above

36) 1 above (L4Ø)

37) A2Ø48,L85,use in G

38) G above

39) A2Ø48,L171,use in H

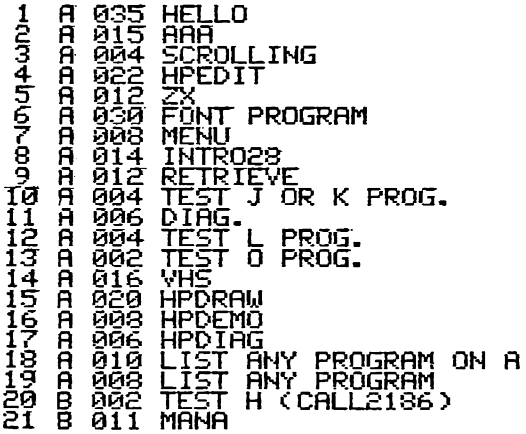
40) A2Ø48,L77,use to replace TEST C in G of 28A,**this DRAW won't work,needs XDRAW like TEST C**

41) MANC,A23Ø4,L1646,delay:7Øhi,255lo,7B shapes,21HX4W,step 2,seq.,use in D

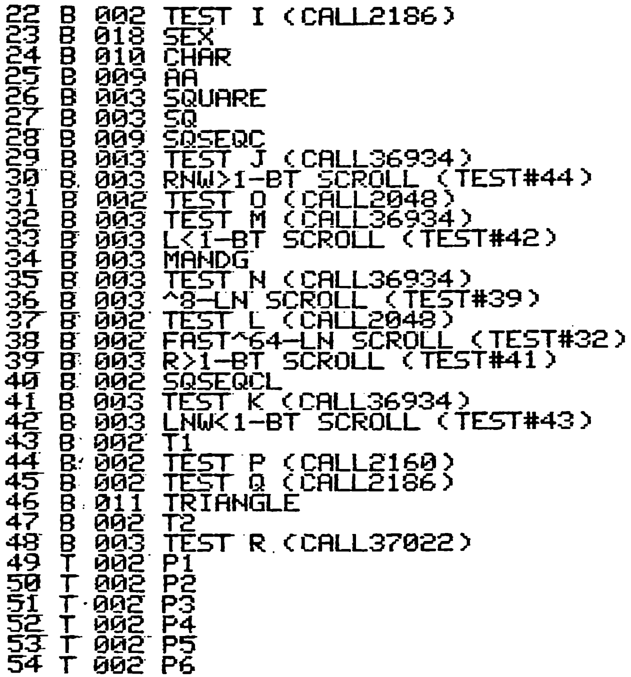
Now we'll look at the menu on 28B

1) create block-shape sequences from other block-shapes automatically
2) animation with vert. XDRAW movement or hor. shift (block-shapes)
3) 2-page block-shape double-shift animation
4) diagonal vert. plus hor. shift (double-shift hor.) animation
5) vertical shift animation, 1-page
6) examine/edit hplot-shape
7) view or animate hplot-shape
8) draw hplot-shapes
9) hor./vert. animate, hplot-shape
A) diag. animate, hplot-shape
B) Superfont
C) Retrieve (use with Superfont)
D) test scrolling routines in Superfont
E) list any 28B program
F) list any 28A program

Now we'll look at the files on 28B:



```
 1 A Ø35 HELLO
 2 A Ø15 AAA
 3 A ØØ4 SCROLLING
 4 A Ø22 HPEDIT
 5 A Ø12 ZX
 6 A Ø3Ø FONT PROGRAM
 7 A ØØ8 MENU
 8 A Ø14 INTRO28
 9 A Ø12 RETRIEVE
1Ø A ØØ4 TEST J OR K PROG.
11 A ØØ6 DIAG.
12 A ØØ4 TEST L PROG.
13 A ØØ2 TEST O PROG.
14 A Ø16 VHS
15 A Ø2Ø HPDRAW
16 A ØØ8 HPDEMO
17 A ØØ6 HPDIAG
18 A Ø1Ø LIST ANY PROGRAM ON A
19 A ØØ8 LIST ANY PROGRAM
2Ø B ØØ2 TEST H (CALL2186)
21 B Ø11 MANA
```



```
22 B ØØ2 TEST I (CALL2186)
23 B Ø18 SEX
24 B Ø1Ø CHAR
25 B ØØ9 AA
26 B ØØ3 SQUARE
27 B ØØ3 SQ
28 B ØØ9 SQSEQC
29 B ØØ3 TEST J (CALL36934)
3Ø B ØØ3 RNW>1-BT SCROLL (TEST#44)
31 B ØØ2 TEST O (CALL2Ø48)
32 B ØØ3 TEST M (CALL36934)
33 B ØØ3 L<1-BT SCROLL (TEST#42)
34 B ØØ3 MANDG
35 B ØØ3 TEST N (CALL36934)
36 B ØØ3 ^8-LN SCROLL (TEST#39)
37 B ØØ2 TEST L (CALL2Ø48)
38 B ØØ2 FAST^64-LN SCROLL (TEST#32)
39 B ØØ3 R>1-BT SCROLL (TEST#41)
4Ø B ØØ2 SQSEQCL
41 B ØØ3 TEST K (CALL36934)
42 B ØØ3 LNW<1-BT SCROLL (TEST#43)
43 B ØØ2 T1
44 B ØØ2 TEST P (CALL216Ø)
45 B ØØ2 TEST Q (CALL2186)
46 B Ø11 TRIANGLE
47 B ØØ2 T2
48 B ØØ3 TEST R (CALL37Ø22)
49 T ØØ2 P1
5Ø T ØØ2 P2
51 T ØØ2 P3
52 T ØØ2 P4
53 T ØØ2 P5
54 T ØØ2 P6
```

1) Hello
2) 1 above
3) D above
4) 6 above
5) 2 above
6) B above
7) Menu
8) introduction, using CHAR
9) C above
10) 3 above
11) 4 above
12) 5 above
13) 6 above
14) 7 above
15) 8 above
16) 9 above
17) A above
18) F above
19) E above
20) A2Ø48,L224,use in 1 or 2
21) MANA,A23Ø4,L237Ø,1ØB shapes,#5 is 15X13 flying saucer,21HX2W,1 step,seq.,use in 2,3,4,5
22) A2Ø48,L233,use in 2

23) SEX,A23Ø4,L4Ø96,16B shapes,#1+#2:27X4,the rest 27HX5W,3-9 ♀ seq,1Ø-16 ♂ seq.,1 step,use in D of 28A

24) CHAR,A36Ø96,L2256,52 V shapes, use in introduction

25) AA,A23Ø4,L1792,7B shapes,16HX5W,step 1,seq.,use in D of 28A

26) SQUARE,A4Ø96,L425,1V shape (white),use in A of 28A

27) SQ,A23Ø4,L421,color=2 for #1 and 3 for #2,2B shapes,32HX4W,use in 2,3,4,5 and D of 28A

28) SQSEQC,A23Ø4,L1792,7B shapes,32HX6W,2 steps,seq.,use in 2,3,4,5 and D of 28A

29) A36864,L292,use in 3

30) A$8ØØ,L$1Ø9,use in D

31) A2Ø48,L82,use in 6,7,8

32) A36864,L333,use in 4

33) A$8ØØ,L$1Ø9,use in D

34) MANDG,A23Ø4,L28Ø,1B shape (2 long),39HX6W,test shape for hor. or vert. shift,use in 2,3,4,5

35) A36864,L333,use in 4

36) A$8ØØ,L$115,use in D

37) A2Ø48,L23Ø,use in 4 or 5

38) A$8ØØ,L$BØ,use in D

39) A$8ØØ,L$1Ø9,use in D

40) SQSEQCL,A23Ø4,L231,1B shape,32HX6W,2 steps

41) A36864,L3ØØ,use in 3

42) A$8ØØ,L$1Ø9,use in D

43) T1,A23Ø4,L28 (small plane),1H (hplot) shape, ≈ Ø,8Ø,use in 9,1Ø

44) A2Ø48,L144,use in 9

45) A2Ø48,L243,use in 9

46) TRIANGLE,A23Ø4,L2317,1ØH shapes, ≈ 14Ø,15Ø,seq 1-1Ø,use in 7,9,1Ø

47) T2,A23Ø4,L28 (big plane),1H shape, ≈ 255,8Ø,use in 9,1Ø

48) A36864,L263,use in A

49) gets Superfont set for RNW>1-BT SCROLL (TEST#44)

50) gets Superfont set for L<1-BT SCROLL (TEST#42)

51) gets Superfont set for FAST∧64-LN SCROLL (TEST#32)

52) gets Superfont set for ∧8-LN SCROLL (TEST#39)

53) gets Superfont set for R>1-BT SCROLL (TEST#41)

54) gets Superfont set for LNW<1-BT SCROLL (TEST#43)

Menu for 28C:

1) write, record, play, see music, done with normal tone routine

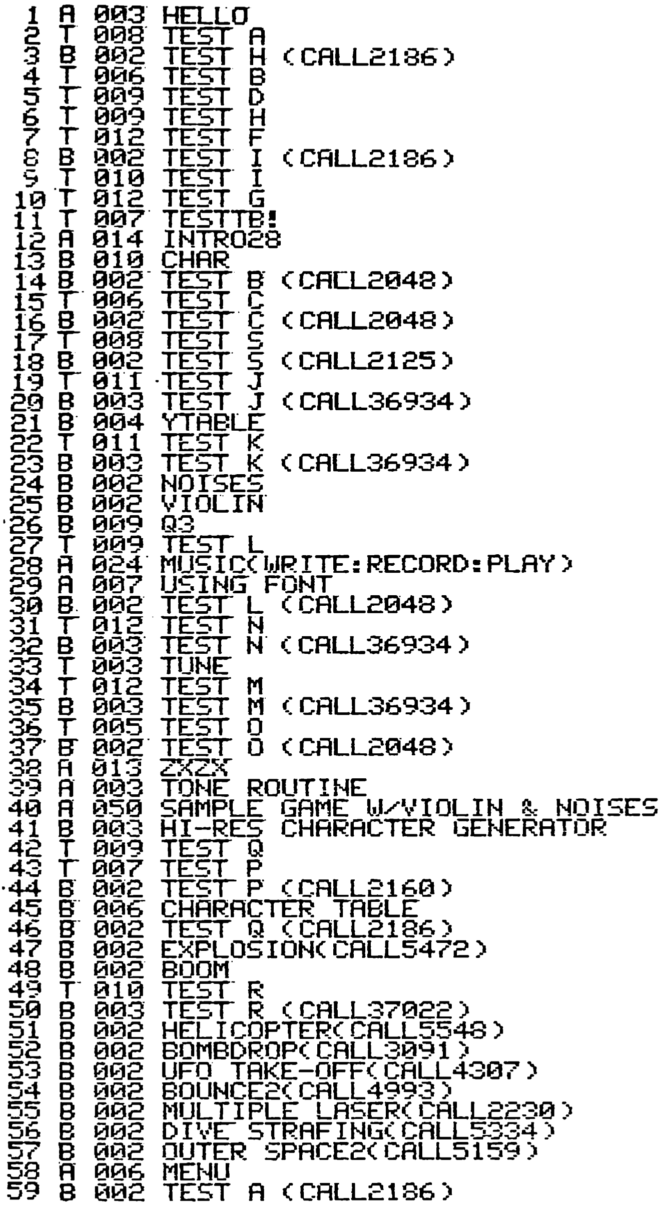
2) sample game shows use of VIOLIN, NOISES, EXPLOSION (CALL5472), BOOM for vector shape animation

3) tone routine with Close Encounters Theme

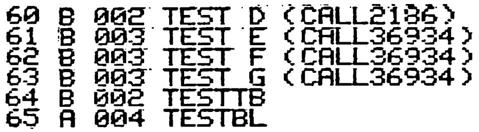
4) YTABLE test, using TEST E (CALL36934) and TESTBL

5) Convert block-shapes to vector shapes

Now let's see the files on 28C:



```
 1 A 003 HELLO
 2 T 008 TEST A
 3 B 002 TEST H (CALL2186)
 4 T 006 TEST B
 5 T 009 TEST D
 6 T 009 TEST H
 7 T 012 TEST F
 8 B 002 TEST I (CALL2186)
 9 T 010 TEST I
10 T 012 TEST G
11 T 007 TESTTB!
12 A 014 INTRO28
13 B 010 CHAR
14 B 002 TEST B (CALL2048)
15 T 006 TEST C
16 B 002 TEST C (CALL2048)
17 T 008 TEST S
18 B 002 TEST S (CALL2125)
19 T 011 TEST J
20 B 003 TEST J (CALL36934)
21 B 004 YTABLE
22 T 011 TEST K
23 B 003 TEST K (CALL36934)
24 B 002 NOISES
25 B 002 VIOLIN
26 B 009 Q3
27 T 009 TEST L
28 A 024 MUSIC(WRITE:RECORD:PLAY)
29 A 007 USING FONT
30 B 002 TEST L (CALL2048)
31 T 012 TEST N
32 B 003 TEST N (CALL36934)
33 T 003 TUNE
34 T 012 TEST M
35 B 003 TEST M (CALL36934)
36 T 005 TEST O
37 B 002 TEST O (CALL2048)
38 A 013 ZXZX
39 A 003 TONE ROUTINE
40 A 050 SAMPLE GAME W/VIOLIN & NOISES
41 B 003 HI-RES CHARACTER GENERATOR
42 T 009 TEST Q
43 T 007 TEST P
44 B 002 TEST P (CALL2160)
45 B 006 CHARACTER TABLE
46 B 002 TEST Q (CALL2186)
47 B 002 EXPLOSION(CALL5472)
48 B 002 BOOM
49 T 010 TEST R
50 B 003 TEST R (CALL37022)
51 B 002 HELICOPTER(CALL5548)
52 B 002 BOMBDROP(CALL3091)
53 B 002 UFO TAKE-OFF(CALL4307)
54 B 002 BOUNCE2(CALL4993)
55 B 002 MULTIPLE LASER(CALL2230)
56 B 002 DIVE STRAFING(CALL5334)
57 B 002 OUTER SPACE2(CALL5159)
58 A 006 MENU
59 B 002 TEST A (CALL2186)
```



```
6Ø B ØØ2 TEST D (CALL2186)
61 B ØØ3 TEST E (CALL36934)
62 B ØØ3 TEST F (CALL36934)
63 B ØØ3 TEST G (CALL36934)
64 B ØØ2 TESTTB
65 A ØØ4 TESTBL
```

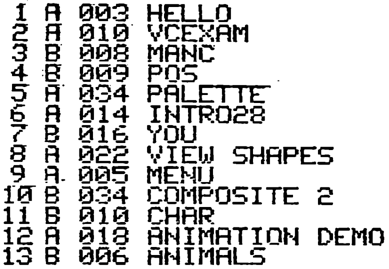
1) Hello
2) use in F of 28A
3) A2Ø48,L224,use in 1 or 2 of 28B
4) don't use, it tries to DRAW rather than XDRAW if used in G of 28A
5) A2Ø48,L21Ø,use in C of 28A
6) use in 1 or 2 of 28B
7) use in D of 28A
8) A2Ø48,L233,use in 2 of 28B
9) use in 2 of 28B
10) use in D of 28A
11) use in 1,3,4,5,6,A of 28A
12) introduction, uses CHAR
13) CHAR,A36Ø96,L2256,52V shapes, use in intro.
14) don't use, it tries to DRAW rather than XDRAW if used in G of 28A
15) use in G of 28A
16) A2Ø48,L85,use in G of 28A
17) use in H of 28A
18) A2Ø48,L171,use in 3 of 28B
19) use in 3 of 28B
20) A36864,L292,use in 3 of 28B
21) YTABLE,A$1DØØ,L$28Ø
22) use in 3 of 28B
23) A36864,L3ØØ,in use in 3 of 28B
24) NOISES,A5625,L114,use in 2 of 28C
25) VIOLIN,A7424,L66,use in 2 of 28C
26) Q3,A23Ø4,L1792 (saucer),7B shapes,18HX5W,step 1,seq.,4of 28C
27) use in 4 or 5 of 28B
28) 1 above
29) USING FONT-test out Font routine usage
30) A2Ø48,L23Ø,use in 4 or 5 of 28B
31) use in 4 of 28B
32) A36864,L333,use in 4 of 28B
33) tune in a text file,used in 1 of 28C
34) use in 4 of 28B
35) A36864,L333,use in 4 of 28B
36) use in 6,7,8 of 28B
37) A2Ø48,L82,use in 6,7,8 of 28B
38) 5 above
39) tone routine
40) 2 above
41) A7168,L256,use in 2 of 28C
42) use in 9 of 28B
43) use in 9 of 28B

44) A2Ø48,L144,use in 9 of 28B
45) A6144,L1Ø24,use in 2 of 28C
46) A2Ø48,L243,use in 9 of 28B
47) A5472,L$4B,use in 2 of 28B
48) BOOM,A37475,L243,5V shapes, vector shape seq. 1-5
49) use in A of 28B
50) A36864,L263,use in A of 28B
51) A$15AC,L$4C
52) A$C13,L$31
53) A$1ØD3,L$BØ
54) A$1381,L$A5
55) A$8B6,L$2C
56) A$14D6,L$89
57) A$1427,L$AE
58) Menu
59) A2Ø48,L189,use in F of 28A
60) A2Ø48,L21Ø,use in C of 28A
61) A36864,L288,use in B of 28A
62) A36864,L324,use in D of 28A
63) A36864,L342,use in D of 28A
64) use in 1,3,4,5,6,A in 28A
65) 4 above

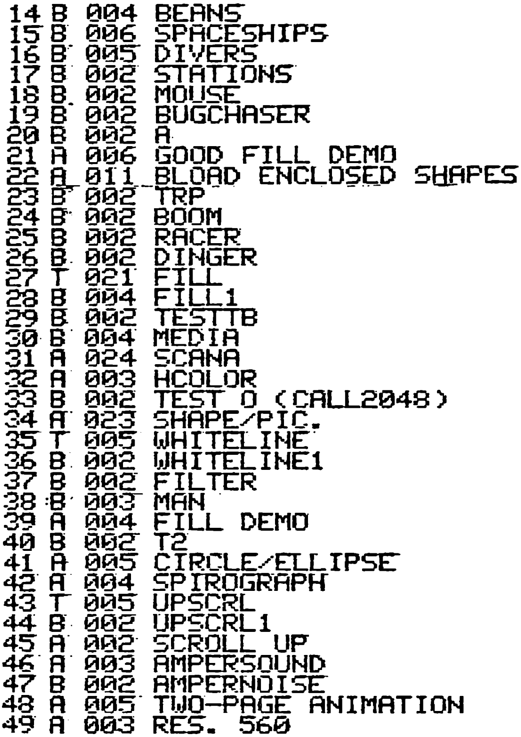
Menu for 28D:

1) Bload Enclosed Shapes
2) Examine Vector Shapes
3) View Shapes
4) Fill with hundreds of colors, paint, fix white lines
5) Vector animation demo
6) Scan/save block-shapes
7) Demo about the 4 adjacent bytes that make up the "color bytes" and what the color bytes are when certain colors are on the screen.
8) Create pictures from shapes; color filtering or other manipulation
9) A demo of how to do automatic filling of shapes and pictures with colors, including a demo of how **not** to fill colors
A) A good demo of color fill --- list it out and see how it's done

Now let's see what files are on 28D:



```
 1 A ØØ3 HELLO
 2 A Ø1Ø VCEXAM
 3 B ØØ8 MANC
 4 B ØØ9 POS
 5 A Ø34 PALETTE
 6 A Ø14 INTRO28
 7 B Ø16 YOU
 8 A Ø22 VIEW SHAPES
 9 A ØØ5 MENU
1Ø B Ø34 COMPOSITE 2
11 B Ø1Ø CHAR
12 A Ø18 ANIMATION DEMO
13 B ØØ6 ANIMALS
```



```
14 B 004 BEANS
15 B 006 SPACESHIPS
16 B 005 DIVERS
17 B 002 STATIONS
18 B 002 MOUSE
19 B 002 BUGCHASER
20 B 002 A
21 A 006 GOOD FILL DEMO
22 A 011 BLOAD ENCLOSED SHAPES
23 B 002 TRP
24 B 002 BOOM
25 B 002 RACER
26 B 002 DINGER
27 T 021 FILL
28 B 004 FILL1
29 B 002 TESTTB
30 B 004 MEDIA
31 A 024 SCANA
32 A 003 HCOLOR
33 B 002 TEST 0 (CALL2048)
34 A 023 SHAPE/PIC.
35 T 005 WHITELINE
36 B 002 WHITELINE1
37 B 002 FILTER
38 B 003 MAN
39 A 004 FILL DEMO
40 B 002 T2
41 A 005 CIRCLE/ELLIPSE
42 A 004 SPIROGRAPH
43 T 005 UPSCRL
44 B 002 UPSCRL1
45 A 002 SCROLL UP
46 A 003 AMPERSOUND
47 B 002 AMPERNOISE
48 A 005 TWO-PAGE ANIMATION
49 A 003 RES. 560
```

1) Hello

2) 2 above

3) MANC,A2304,L1646,delay hi 70,lo 255,7B shapes,21HX4W,step 2,seq.,use in 2 or 6

4) POS,A2110,L2830,126 (actually 14 done) V shapes about yoga,use in 2,4,6,8

5) 4 above

6) introduction,uses CHAR

7) YOU,A2336,L3700,13V shapes (12 done),monsters,use in 2,4,6,8

8) 3 above

9) Menu

10) COMPOSITE 2,34-sector picture that BLOADs in page one, use in 4 and 8

11) CHAR,A36096,L2256,52 vector shapes

12) 5 above

13) ANIMALS,A$8FFF,L$44E,22 V shapes (last 6 gone)

14) BEANS,A$9000,L$1FF,9 V shapes

15) SPACESHIPS,A$9000,L$47B,50 V shapes, (last 18 gone)

16) DIVERS,A$9000,L$3E5,15 V shapes

17) STATIONS,A$9000,L$93,3 V shapes

18) MOUSE,A$9ØØØ,L$BØ,4 V shapes
19) BUGCHASER,A$9ØØØ,L$DB,3 V shapes
20) A,A3936,L49,3 V shapes (try DRAW **and** XDRAW in 3 of 28D)
21) A above
22) 1 above
23) TRP,A$925C,L$F,1 small torpedo shape,V
24) BOOM,A$9263,L$F3,5 V shapes,seq. 1-5
25) RACER,A$9Ø95,L$1B,1 racer shape,V
26) DINGER,A$145Ø,L$E,1 small square shape,V
27) FILL,source for FILL1,use in 4
28) FILL1,A$9ØØØ,L$32Ø,use in 4
29) TESTTB,A2Ø48,L14Ø,use in 4,6,8
30) MEDIA,A$9ØØØ,L$267,2Ø V shapes
31) 6 above
32) 7 above
33) TEST O (CALL2Ø48),A2Ø48,L82,use in 9
34) 8 above
35) WHITELINE,source for WHITELINE1,use in 4,8,9
36) WHITELINE1,A2Ø48,L111,use in 4,8,9
37) FILTER,mach. lang. color filter routine
38) MAN,A$93Ø,L$14B,8 V shapes
39) 9 above
40) T2,A23Ø4,L28,1 H shape,use in 9
41) Circle or ellipse drawing
42) Spirograph/polar coordinates graphics
43) UPSCRL,source for UPSCRL1,use in SCROLL UP
44) UPSCRL1,A$8ØØ,L$6Ø,one line at a time scrolling,binary,use in SCROLL UP
45) SCROLL UP,Applesoft driver for UPSCRL1,1-line scroll-up
46) ampersand controlled Applesoft sound routine demo
47) ampersand controlled binary sound routine
48) 2-page animation demo (fundamentals)
49) 56Ø-pt. resolution demo

HOW TO USE THE PROGRAMS ON THESE DISKS

There are several types of files on these four disks. One type is a shape table, such as: B009 Q3

This a binary file in which a small block of memory containing one or more Block-Shapes is found. The first byte of the address at which Q3 is stored does not contain the number of shapes in the file, as with Vector Shapes. Also, there's no index, and no 00 to represent the end-of-shape byte. Not all binary files contain Block-Shapes. Binary files on these disks will also contain:

1) Hplot-Shapes
2) Vector Shapes
3) Machine Language Graphics Programs
4) Machine Language Scrolling Routines
5) Machine Language Sound Routines
6) Hi-Res Character Generator
7) Hi-Res Character Table
8) YTABLE, for indexed vertical addresses

Another type of file is an Applesoft program such as: A020 HPDRAW:

This is an Applesoft program that lets you draw Hplot-Shape tables and save them. Other Applesoft programs on these disks let you save, draw, examine, edit, animate all types of shapes, run Superfont, sample game, music program, tone routine, choose from menus, list out programs, create or retrieve/view text file array shapes.

Here's another type of file: T008 TEST A

This is a LISA-execable test file containing LISA source codes for the machine language program: B002 TEST A (CALL 2186)

The way you'll use these LISA files is by doing a CTRL D EXEC (file name) in LISA, and then hit A to assemble them. Now you may change/append them or simply examine them.

Another test file: T002 QQ.

This file holds a shape in test file array form. It was "drawn" by program 7 on disk 28A. It can be viewed with program 8.

Another test file: T002 P1

This file holds the **name** of a scrolling program BLOADED by: A 028 FONT PROGRAM.

If you need the address of any binary file you've just BLOADED, then type: ? PEEK(43634) + PEEK(43635) * 256.

If you need the length of this file, type: ? PEEK(43616) + PEEK(43617) * 256.

(These commands are for use with the unprotected 28C or 28D disks in immediate or deferred mode or for shapes or programs you create on your own unprotected disks.)

These disks are written in either DOS 3.3 or DOS 3.2. If you have the 3.2 version and try to muffin it, things will mess up because disks 28A and 28B are protected. Get the correct DOS version from your dealer or from us to begin with, or for $5 exchange one DOS version for the other (send us the old version and a note and $5). By the way, the way you get from one protected program disk to the other (28A/28B) is to switch disks before you leave a program. When programs are exited they go to MENU, but MENU is on 28A **and** 28B, so it's okay to switch disks here. Now, 28C and 28D are unprotected so you can't go from them to 28A and 28B without rebooting, or from 28A or 28B to 28C or 28D.

Let's look at the ways you'll be using these disks. First, of course, you'll want to go through the manual and try all the programs on the disk. Once you're familiar with what the various types of shapes, sounds, and routines are all about, you may list and study programs that represent procedures and/or shape types you'd like to learn more about. Once you understand the program, you may use it in your own programs like so:

1) Figure out how much of the program you'll need

2) Type and save that in your own program. (This refers to the Applesoft programs that function as drivers to machine language drawing programs, such as ASMBLSEQ on disk 28A. A **driver** is a program that runs another program or output device or sends input to it.)

3) BLOAD the correct binary graphics routine off disk 28C and BSAVE it onto your personal disk (TEST C (CALL 2Ø48) would go with ASMBLSEQ, the 19th file on disk 28A). The proper addresses and lengths are written in this manual, and can also be found with the PEEKS above.

4) If you haven't already created a Block-Shape sequence table for your program, either do so or BLOAD Q3 off 28C and BSAVE it on your personal disk (if such a sequence is needed).

5) Make sure your HELLO program does POKE 1Ø3, 1:POKE 1Ø4, 64:POKE 16384,Ø **before** it LOADS your graphics program, if it's a **1-page** Hi-res program (using only HGR). If it's **2-page flipping** use POKE 1Ø3, 1:POKE 1Ø4, 96: POKE 24576,Ø. This starts your program higher than your graphics tables and the hi-res pages.

6) Make sure your program also:

(A) BLOADS all necessary shapes

(B) Uses shapes that were saved with the correct addresses and lengths

(C) BLOADS any character generators and tables you'll be using

(D) BLOADS any machine language graphics routines you'll be using

(E) Sets Himem to 36864 if tables or machine language programs are BLOADED above 36864 ($9ØØØ).

(F) Correctly handles steps, coordinates, variables, colors, travel direction

(G) POKES 234 ($EA) (NOP) into **EOR** ($51) and (HBASL),Y ($26) lines in machine language routines if you prefer DRAW over XDRAW. (In BLOCK-SHAPES it's EORing that causes XDRAW instead of DRAW, since in EOR a bit gets turned on only if it's different than the one it is drawn on top of.) (Be forwarned that block shapes **will not erase** if you dump the EOR line!)

(H) Uses ONERRGOTO routines

(I) Employs realistic lengths in delay loop variables

(J) BLOADS YTABLE at $1D00(length is $280) if you'll be substituting table look-up for HPOSN in finding addresses of vertical coordinates, and also POKES in the proper YTABLE addresses at 222, 223, 206, 207, 30, 31 (dec.), and replaces JSR $F411 (HPOSN) with JSR $320 (YTABLE vertical address locator routine) (JSR is left as is, and POKE 32 for low byte and 3 for high). Also, POKE in proper $320 vertical locator routine.

(K) Calls the correct graphics routine number

(L) Has delimiters/boundaries or conditions that stop action from achieving infinite loop

(M) Uses correct 232 and 233 (dec.) POKES if shapes are vector, and specifies HCOLOR, ROT and SCALE.

SHAPES AND OTHER MYSTERIES 7

What is a shape? A shape is a figure of some kind seen on a CRT (the video monitor). It may be white on black, black on white, or colored. It may be hi-res or low-res, or even text. We'll be concentrating on hi-res for now.

On the screen the figure of a shape is composed of a special configuration of bits turned on (or off if you're drawing in black on a white background). The shape may include only the figure itself, but the bytes in the shape table that are responsible for drawing it may include lots of invisible dots as well. The actual bytes in a shape may include as little as 5% visible dots. This will often occur in unfilled block-shape figures. All dots in hplot-shapes are visible. A vector shape's dots may either all be visible or only part of them may be visible. Invisible dots are included in the shape as moving vectors in which no plotting is done.

In memory, a shape may be several different things. It may be mere #s in a BASIC DATA statement. Or it may be numbers in an array that is either in a BASIC program or saved in a sequential text file and read into a BASIC program with DOS commands. This system contains programs to do the latter: #7 in disk 28A will scan a vector shape and form a text-file from the bytes it reads (a block-shape configuration); #8 will READ a saved text file array shape from a text file and display it, and #9 will display it upside down.

This text file stuff is not usually practical or the best answer. It is easy to understand, though, so this is the main reason it was included in this system. What these text file shapes do is represent a block of memory saved as an array in a text file. This block of memory includes all the bytes necessary to include the entire shape (plus some extras, usually).

(each of the 7 dots in a byte is a bit)

		Byte #0	Byte #1	
address$2000	0	○○○○○○●	●○○○○○○	address$2001
$2400	1	○○○○●●○	○●●○○○○	$2401
$2800	2	○○●●○○○	○○○●●○○	$2801
$2C00	3	●●●●●●●	●●●●●●●	$2C01
$3000	4	○●●○○○○	○○○○●●○	$3001
$3400	5	○○○●●●○	○●●●○○○	$3401
$3800	6	○○○○○○●	●○○○○○○	$3801
$3C00	7	○○○○○○○	○○○○○○○	$3C01

The above figure shows a very small flying saucer. It's the type of shape arcade. games like to move around swiftly to provide action/adventure. It is 2 bytes wide and 8 lines high. This makes it a 16-byte block-shape. One can load 16 bytes into memory extremely fast, especially from a mach. lang. routine. There are plenty of such mach. lang. routines in this graphics package.

If you were to load a shape like this from a text file array shape and display it with program 8 of 28A, you wouldn't need to know any data, you'd just do it. If you were to load such a shape from a binary block-shape table, you'd need to know:

1) the shape # (1-23)
2) the VT (top or highest vertical coordinate)
3) the VB (bottom or lowest vertical coordinate)
4) the HR (right-most horizontal coordinate)
5) the HL (left-most horizontal coordinate)

For the shape in the figure above the coordinates are (it's OK to add a # to VT and VB, and/or HR and HL):

VT=0
VB=7
HR=1
HL=0

And the shape # would be whatever one it had been given when it was stored as a shape; let's call it shape #1. Some of this will seem unclear right now, but don't worry, before I'm through you'll know more about shapes than Hugh Hefner.

Look at figure #1 again. Notice that less than half the bytes in this block-shape are "on". The rest are merely part of the rectangular space the flying saucer happened to be in. All 16 of the bytes that make up this shape are shape-bytes, but not all the bytes are necessary parts of the shape itself. The first 2 bytes are not even part of the shape at all --- they could just as well have been skipped.

That brings up a point: in block-shapes, the less excess bytes are used, the less time it will take for a block-shape to load onto the hi-res screen and the faster it can be animated.

Again consult the figure. It could be put there by text file array READING and then loading the bytes from the text file into memory, or it could be put there by moving a block of memory from a binary block-shape table 16 bytes long into the addresses $2000, $2001, $2400, $2401, $2800, $2801, $2000, $2001, $3000, $3001, $3400, $3401, $3800, $3801, $3000, $3001.

The reading for a block-shape would start at HR, VB (2nd byte (byte #1) of the 8th line (line #7)), or $3001, and read the first block-shape table byte into it. The next byte in memory (in the block-shape table) would be read into $3000, the next $3801, then $3800, and so on until the 16th byte in the shape table would be read into the address $2000 (HEX).

The difference between reading the shape from a block-shape table and from a text-file is that the text file inputs bytes until a RETURN signals end-of-field, and then it goes on to the next field, until all 16 are read in. They are placed on hi-res page one by POKES, just as they are scanned **from** page one ($2000-$3FFF) and put into a text file in the first place by PEEKS followed by OPEN, WRITE, PRINT (18 of these), CLOSE. Also, they are read from line #0, byte #0 to line #7, byte #1, just the opposite of block-shape scanning and drawing. (The reason there are 18 PRINTS above rather than 16 is that the first 2 inputs into the text file are **width** and **height**. This information allows the creation of a 2-dimensional array. NUM (X,Y) would be a general element of this array. NUM (2, 3) in the above figure would be $2801's byte.

With a binary block-shape table, the data array is there, just as it is in a text file block-shape array, but with binary files there is nothing there but block-shape bytes. There are no shape table #s or height or width data of any kind. So to use a block-shape table you need to know what's in each shape # (from 1-23 is possible shape #) and what the width and height is. It follows, then, that when you create shapes, you'll be smart to jot down this information.

There's a funny thing about these block-shapes and I may as well tell you now so the laughter may commence: even though you've already learned that the above shape is an array 2 wide by 8 tall, when you're asked for the width and height in a shape-viewing program you'd give a width of 1 and a height of 7!

Before you get the idea that I'm off my rocker let me explain. If you get really technical about it, when you deal with block-shapes the question "what is the shape's height?" doesn't mean "how many lines tall is it?". Instead, it means "exactly how far is it from the lowest byte to the highest byte in this shape?". Well, from 0 to 7 is 7, not 8. To check it, subtract 0 from 7. You'll get a remainder of 7. The same with the width: 1 minus 0 is 1 so we say that shape is "1 wide", meaning "the left byte is only 1 away from the right byte". This means that it would be okay to enter coordinates of VB=119, VT=112, HR=33, HL=32. Or if asked for height and width you'd say 7 and 1. You see, from line 112 to (and including) line 119 is 8 lines, and from line-byte 32 to (and including) line-byte 33 is 2 bytes. So the **true** dimensions of 2 wd. x 8 ht. are the end results. The secret here is that the block limits you give **are**

used inclusively. And 7 and 1, as height and width, get 1 added to them in the course of their utilization, meaning that the program recognizes that these numbers refer to how far parameters are from each other, rather than actual heights and widths. Neat, huh?

You may be wondering: "Since most of the shape's bits are "off", wouldn't it be better to use vector shape drawing rather than block-shapes?" Well, it would take about 26 vector-plot bytes to draw this figure, and it would have to be put through Applesoft DRAW or XDRAW routines. So it would be slower and take up more memory. But if you feel you need to DRAW and XDRAW and especially if you need to ROTATE and SCALE, there's a lot to be said for vector shape methods.

Block-shapes do either normal byte loading (STA) onto the hi-res page or EOR and then STA, which means the bytes are logically exclusive-ORed with what's on the hi-res page before they're loaded onto it. The first method is comparable to DRAW, the EOR method is comparable to XDRAW. A block-shape routine needs only one line added to get it to go from STA to EOR and STA, the XDRAWing. Hplot-shapes are no good at XDRAWing, however.

XDRAW means leaving the background as you found it once you've left (which is much easier to handle in black and white than in color) and it means having 2 whites make a black, so that where shapes are superimposed there will be blackness. Either-or means that 1 is the result if the 2 compared bits of the 2 compared bytes are different (one black, but one white) and Ø is the result if they are the same (so 2 "1"s would be 2 whites forming a black.

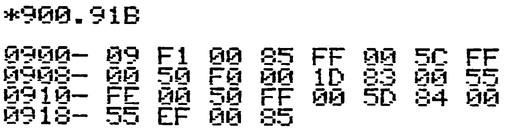
Hplot would not be a good shape method here since hplotting is only economical if there are mostly long (or few) straight lines in a shape. An hplot-shape table is in the same place (starts with shape #1 at $9ØØ, #2 at $AØØ, up to #23 at $1FØØ) as a block-shape table, but is done very differently.

You needn't give coordinates --- for an hplot-shape you need give only 1 thing (unless you wish to give a different **color** to its lines) and that's its shape #. The data in an hplot-shape represent turning points in the shape --- they're the coordinates of the points where the lines join. Each point is defined by 3 bytes:

1) the horizontal coordinate's low byte (Ø-255)
2) the horizontal coordinate's high byte (Ø-1)
3) the vertical coordinate's byte (Ø-191)

Here's the way a simple, but large, shape of the HPLOT type would look in a binary file: (There are no shape-bytes here, just point-coordinates.)

PRINTOUT #1



```
*9ØØ.91B

Ø9ØØ- Ø9 F1 ØØ 85 FF ØØ 5C FF
Ø9Ø8- ØØ 5Ø FØ ØØ 1D 83 ØØ 55
Ø91Ø- FE ØØ 5Ø FF ØØ 5D 84 ØØ
Ø918- 55 EF ØØ 85
```

The first byte of an hplot-shape is different from either a vector shape or a block-shape. It's a number that represents the **# of points in the shape.** If the shape is a closed shape, like an octagon, there will be one point that is counted twice, since the last line draws back to the staring point (or one or 2 dots from that). There are 28 bytes in the shape, the first being the # of points and after that are 9 groups of 3 bytes (hor. lo., hor. hi., vert.).

The shape #, which normally gets passed at $7 in most programs in our system, is determined by what page the shape is stored at. What is a "page"? It's 256 consecutive bytes of memory sharing a common high-order byte in their addresses, such as page 9 which is $900 to $9FF. What is a "high-order byte"? Well, if you were to see 00 09 in the monitor and were sure it represented **an address** rather than data or something else, it would mean $900 (dec. 2304). The low byte is normally before the high byte, as that's the way the computer likes to deal with it. The high byte's # means **that times 256,** while the low byte's # means **that times one.** It's equivalent to the third digit in the decimal system meaning **that times 100.** The first place is one's place in either hex (base 16) or dec (base 10). The second place is **times 16** in hex and **times 10** in dec. The third place is **times 256** (16^2) in hex and **times 100** in dec, and the fourth place is **times 4096** (16^3) in hex and **times 1000** in dec. Many of you know this --- some of you don't.

So the shape # is dependent upon which page the shape is stored at. There are 23 pages from $900 to $1F00, so there's room for 23 shapes. The limit is 256 bytes per shape, and unlike vector shapes, no end 00 byte or index is required. Block or hplot-shapes may only be a few bytes long, but still they're allotted 256 bytes. It's okay to give them more than this --- even up to 5888 bytes, which is about 2/3 of the hi-res screen. This last # is the distance from $900 (2304 dec.) to $1FFF (8191 dec.). (There are many other ways to handle shape #/mem. addr. relationships; this is merely Fudgie's way.)

Why do both block and hplot-shapes start at $900 and why **don't** vector shapes start there? Simple. Free memory starts at $800. But many of our machine language graphics routines are stored from $800 to $8FF (longer ones are from $9000 to $9150, approximately), so we started our shape tables at $900. When using graphics programs it's often best to POKE 103,1: POKE 104,64: POKE 16384,0 in HELLO to start the programs above $4000 and avoid conflict with page one graphics, or, for dual-page "flipping" programs to POKE 103,1: POKE 104,96: POKE 24576,0 to avoid conflict with **both** hi-res pages by starting the program above $6000. "Page" here actually means all the pages from $2000 to $4000 or from $4000 to $6000, the latter representing hi-res "page" 2 and the former representing hi-res "page" one.

Even though hplot-shapes are like block-shapes in terms of shape # assignment and placement in memory, a block-shape has no special first byte --- all bytes are simply raw graphics bytes to be loaded into memory according to the configuration you've jotted down somewhere (shape #, height, width).

Now, the reason that all vector shapes don't start at $900 is that they start anywhere from $800 to $9600, **including** $900. The reason each shape is given 256 bytes is that this makes for very fast determining of where to go to get the block-shape or hplot-shape bytes for the specified shape #. By having one shape per page, you need only deal with the high byte of the shape's address. here's how it's done:

```
      LDY #$9       start with high byte of 9 in Y register
      LDX  $7       stick shape # into X
HERE  DEX           subtract 1 from shape #
      CPX #$ØØ      is it Ø yet?
      BEQ THERE     if it's Ø go to THERE
      INY           if it's Ø still, add 1 to Y (high byte)
      JMP HERE      and then loop back to HERE
THERE TYA           now that Y is up to correct shape address
      STA BASH      high byte, put Y into base
                    address high byte, etc.
```

Shapes are usually numbered quite low, so very little looping need be done to determine exactly what is the correct shape base address high byte (remember that the low byte will stay Ø so it is not dealt with --- if you'd like to cram lots more shapes into the 5888 available bytes, all you need to do is either build an index table similar to the ones found in normal vector shape tables, or extend the above routine to include manipulation of low bytes: you could add #$4Ø (64 dec.) 4 times within each page and have room for 92 shapes, not 23).

To recap for a second, for an HPLOT shape you'll need to know the name of the shape table and the # of the shape you want. This will be drawn at a particular place in memory. But this doesn't mean you can't move or animate hplot shapes. Quite the contrary --- they're easy and fun to move/manipulate, and the lines may be of whatever color you choose. The shape will always **start** in the place designated by the point coordinates stored in memory, but after that it's up to you. All the routines you'll need for drawing, scanning, examining, locating, moving, coloring, editing and understanding hplot-shapes will be found on disk 28B. The advantages of hplot-shapes are these:

1) they take up very little memory
2) they're very easy to use
3) they're quick and efficient
4) you may change their color easily
5) they're best for shapes composed of a series of **straight lines**

To recap vector shape info already discussed either here or in your Applesoft manual: Vector shapes are best when you have to rotate shapes and/or change their scales (larger or smaller). They aren't as fast as hplot-shapes or block-shapes, but are still plenty fast enough for decent animation, as long as you don't have too many of them running concurrently. One needs to specify ROT (rotation of Ø to 255) and SCALE (0 to 255) and HCOLOR (0-7) and also shape table address via dec. 232 and 233. ($9000 would be POKE 232,0: POKE 233,144. The 144 is dec. of $90, the high byte of the shape table address, which again is after the low byte of ØØ; 233 is after 232 in memory). When DRAWING one needs to specify DRAW or XDRAW, and you need to give shape # and coordinates, and HGR or HGR2 or both must have happened (init. graphics).

I feel vector shapes are the best way for beginners to start with graphics shapes. To learn more consult both the Applesoft Program Manual and our Super Shape Draw and Animate Package --- the latter contains 2 manuals and 2 disks that make drawing and animation a breeze. The advantages of vector shapes are these:

1) Applesoft is already fully equipped to handle them, with its DRAW and XDRAW routines and its SCALE, ROT, HCOLOR factors.

2) They grow and shrink and rotate and change colors easier than other types of shapes.

3) They're the best shape type for beginners, and are the easiest to use from BASIC.

4) There are many commercially available drawing and animation programs which create/utilize vector shapes.

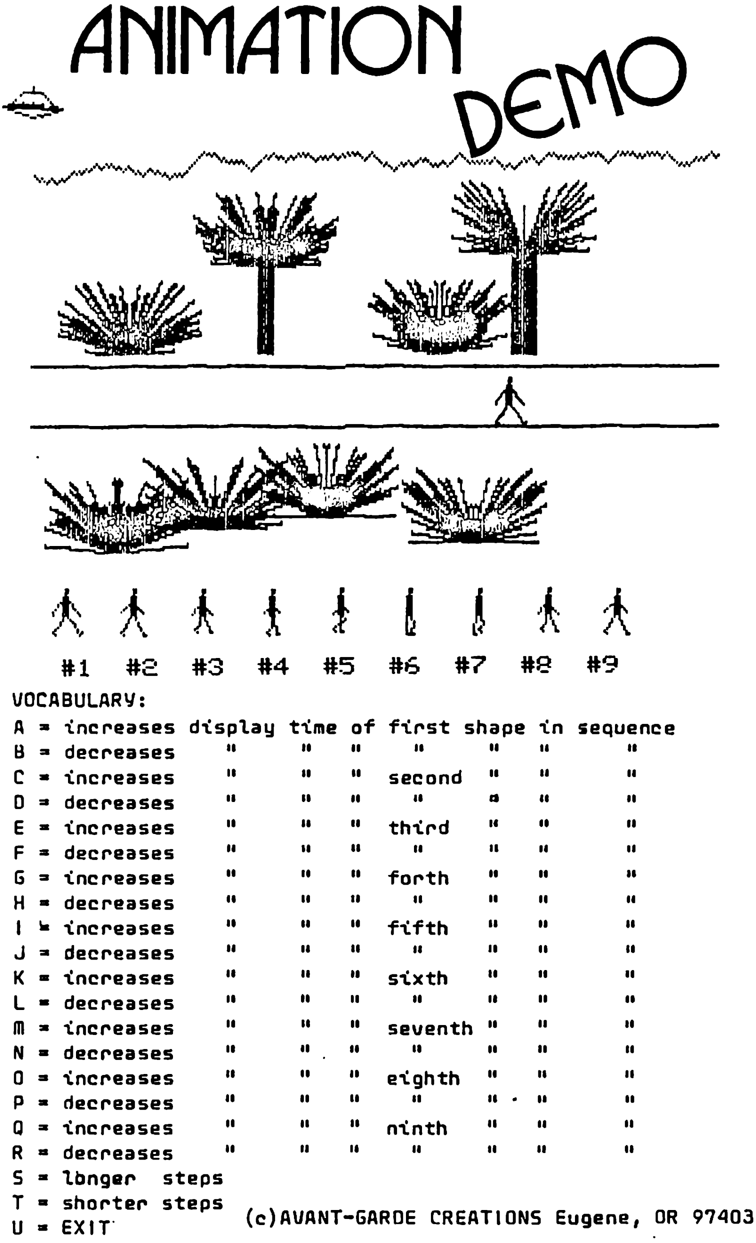
Block-shapes are faster than vector shapes. Let's see why. Each byte in a vector shape table draws at most 2 dots. Each byte in a block-shape table draws 7 dots (whether visible or not). Block-shapes may avoid the time-consuming HPOSN routine (part of DRAW or XDRAW) by using the YTABLE option, which **finds** (by indexing) the hi-res page address of the vertical coordinate in a couple of machine language bytes of time, rather than **calculating** (by logical shifts and loops, etc.) the address like HPOSN does. Vector shapes need to do most of what happens in HPOSN, so even when utilizing YTABLE things get slowed down considerably. Block-shapes can be saved quickly and easily from parts of anything that gets drawn on the screen,including combinations of other shapes of other types. Vector shapes must be drawn a dot at a time (with the exception of the program 5 of 28C, which converts anything on the screen into a vector shape which can be rotated, scaled, etc.) and take a very elaborate routine to create from other shape types. So the advantages of block-shapes are:

1) ease of creation from anything on hi-res screen
2) efficiency and simplicity of storage
3) rapidity of drawing/animation
4) ease of changing back and forth from DRAW to XDRAW
5) compatible with use of YTABLE, for extra speed
6) any # of colors in **each** shape

Shapes of all types not only need to be used individually, they also need to be used as sequences.

On the next page is a sequence from an animation demo on our Super Shape Animate disk. A sequence isn't always in consecutive shape # order --- also, sometimes a shape is used more than once in a sequence. You'll be trying out sequences in program #3 of disk 28A. These must be sequences that move a byte at a time (there are 40 horizontal bytes in Apple graphics) rather than a bit, or they may move more than a byte at a time. This is pretty large stepping, so the programs that move only 1 or more bits at a time may prove to be more useful. (Examples are programs A,B,C,D,G,H on 28A and 1,2,3, 4,5,7,9,A on 28B and 4 on 28C, which uses the YTABLE).

Colored shapes need to move 2 bits at a time to avoid color change, if the movement is horizontal. For vertical shapes this is irrelevant. The reason that colored shapes need to move an even number is that hi-res color graphics really only has 140x192 resolution, not 280X192. (If you get **really** technical, like one Apple Orchard article



```
VOCABULARY:
A = increases display time of first shape in sequence
B = decreases    "     "   "    "     "   "     "
C = increases    "     "   "  second  "   "     "
D = decreases    "     "   "    "     "   "     "
E = increases    "     "   "  third   "   "     "
F = decreases    "     "   "    "     "   "     "
G = increases    "     "   "  forth   "   "     "
H = decreases    "     "   "    "     "   "     "
I = increases    "     "   "  fifth   "   "     "
J = decreases    "     "   "    "     "   "     "
K = increases    "     "   "  sixth   "   "     "
L = decreases    "     "   "    "     "   "     "
M = increases    "     "   "  seventh "   "     "
N = decreases    "     "   "    "     "   "     "
O = increases    "     "   "  eighth  "   "     "
P = decreases    "     "   "    "     "   "     "
Q = increases    "     "   "  ninth   "   "     "
R = decreases    "     "   "    "     "   "     "
S = longer  steps
T = shorter steps
U = EXIT             (c)AVANT-GARDE CREATIONS Eugene, OR 97403
```

did, black and white resolution is 560X192 and color is 140X192 and the reason most Apple users and programs think in 280X192 terms is that it's easier to work with. Routines that take advantage of 560 horizontal resolution tend to be awkward and cumbersome. The Graphics Tablet has 560-dot resolution capacity. More on this later. We'll continue to think in 280x192 resolution terms to simplify things. Anyway, specific colored dots show up right at even numbers only, and others work best with odd, so if you have a blue shape that's showing up well as blue and then move it over 1 bit or dot, presto --- a color change (depending upon the color and tint settings on the monitor)! The disks include animation routines that take this color-change phenomenon into account, and others that you can turn into color-careful programs just by using the proper shape sequences in them (drawn from probably one of the following programs: 4 on 28A, A on 28A, 1 on 28B).

If you're going to do animation, remember that you'll often be moving either 1 or 2 bits per step. This is easy to do with vector shapes since all you do is XDRAW 1 at 40, 50:XDRAW 1 : XDRAW 1 AT 42, 50: XDRAW 1. (The first XDRAW draws, the second erases, the third redraws at a new location), the fourth erases, etc. The reason no coordinates are given for the second (erasing) XDRAW of each of the above 2 movements is that Applesoft remembers the location of the last shape plotted and uses these coordinates again.)

By the way, a program like the above would make dim, flickery shapes, since the screen would be blank as often as it would contain a shape. The way the dimness is corrected is to start program loops in which the old shape is erased and then the new shape is immediately drawn, and after this any calculations, delay loops, keyboard or paddle button reading, or IF—THENS are done before looping back to the quick erase/draw commands. More on this later.

Also, in order to never show a blank screen, even for a tiny fraction of a second from the erasing to the drawing, flipping between hi-res pages is often used. This means your program has less memory and must start after $6000, but the animation quality is worth the sacrifice in many cases. Two other ways to hold down flicker are the use of logical-shift shape-moving, which requires no erasing (but does require page flipping if you're using colors, as well as double-shifting), and fast assembly drawing routines with delays during the shape-visible segment of the routine. Things happen so fast that no erasing is noticeable.

With the block-shape routines included in this sytem, necessary 1 or 2-bit movements in animation are easy enough to do, using the right shape-table sequence tables, but if you try it with the wrong sort of shapes or with only 1 shape, problems may arise. One shape cannot move only 1 bit sideways unless it goes to another shape # in a sequence or goes through either ROL or ROR logical rotation-shift routines. Examine figure #1 again. You could "draw" this shape just as well at VT=1, VB=8, HR=2, HL=1 as you could at 0, 7, 1, 0. This would be a mostly rightwards but slightly downwards move for this block-shape. The width and height would still be actually 2X8, and would still be referred to as 1X7 (for reasons already discussed). But what about moving this shape rightwards **less** than one byte? How can we move it over 1 bit? How do we draw this shape not at 0, 7, 1, 0 or 0, 7, 2, 1 but at 0, 7, 1 1/7, 1/7? The answer? Logical shift, or form it into 6 other additional shapes, each moved over 1 bit, and save all these shapes, including the first one, as a shape table sequence. (There are 2 automatic shape-sequence creators in this system. A on 28A starts with a vector shape and ends with 7 block-shapes, each 1 bit over from the last. 1 on 28B starts with a block-shape and ends up the same as the above program --- it used logical shifting as its magical problem solver.)

The bottom line here is that a block-shape is drawn with HR and HL from 0 to 39 only --- there are only 40 possible horizontal coordinates, since the bytes these block-shapes are composed of are a full seven bits long. (The reason they're not 8 bits long is that the high bit (bit #7, the 8th or left-most bit in memory) is the color bit and is not visible.) (7X40 is 280, the hi-res resolution.) So block-shapes are always in need of overcoming their inherent problem of movements of under one byte. The solutions are fairly obvious and easy, but always keep in mind that you **can't** draw a block-shape at 3 4/7 HL and 4 4/7 HR. It must be even #s. To get that shape to be at that desired screen position, think in terms of shifting it to there or creating a sequence in which the fifth of 7 shapes is at the desired screen position. Look back at figure #1. Obviously you'll have to have a 3-byte-wide block-shape to let a moved over version be created. (You'll be **saying** that it's 2 bytes wide, obviously, for reasons already discussed.)

If you wanted to move it over 1 bit only then color wouldn't work, unless you **desired** a 2-color flicker. For a sequence (of 7) to be in color you'd need the shape to be 4 bytes wide and call it 3 wide, and each shape in the sequence would be 2 dots from the last. The option to move shapes over more than 1 bit at a time in the 2 automatic shape-sequence creation programs is available.

So what if you want walking, talking, running, swimming types of animation where the figure's movements must be represented by 7 or more **different** shapes? I'd recommend using our Super Shape Draw and Animate Package and drawing this sequence as vector shapes, and then entering the present system and creating a block-shape 'different-shape" sequence with program 4 on 28A. You **could** do it by using 2 on 28A to draw the shapes originally as block-shapes (after each drawn shape you'll be sent to 4 on 28A to scan and save the shape), but this might be more awkward to accomplish --- you'd have to be very picky about the precise coordinates of the parts of each of the shapes. Perhaps you'll not find that much of a barrier and you will make your walk-animation sequences this way. All I can say is that I prefer the Super Shape Draw and Animate Package for this purpose. I spent a long time making it the easy-to-use draw/animate package that it is, so I'd be very silly to do walk-animation any other way. I've yet to see an easier-to-use vector shape drawing and animation package.

SHAPE DRAWING

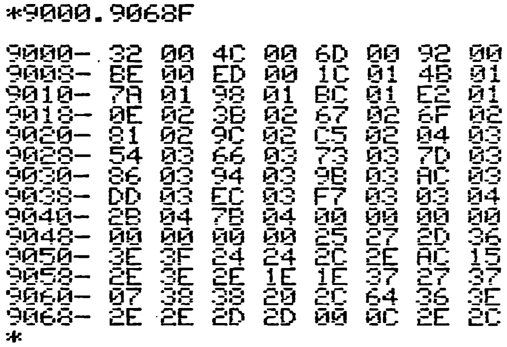
8A VECTOR SHAPES

Here is a vector shape:

PRINTOUT #2

Here is the shape table for this shape: (the shape shown goes from $9Ø4C to $9Ø6C; it's #1)

PRINTOUT #3



```
*9ØØØ.9Ø68F

9ØØØ-  32 ØØ 4C ØØ 6D ØØ 92 ØØ
9ØØ8-  BE ØØ ED ØØ 1C Ø1 4B Ø1
9Ø1Ø-  7A Ø1 98 Ø1 BC Ø1 E2 Ø1
9Ø18-  ØE Ø2 3B Ø2 67 Ø2 6F Ø2
9Ø2Ø-  81 Ø2 9C Ø2 C5 Ø2 Ø4 Ø3
9Ø28-  54 Ø3 66 Ø3 73 Ø3 7D Ø3
9Ø3Ø-  86 Ø3 94 Ø3 9B Ø3 AC Ø3
9Ø38-  DD Ø3 EC Ø3 F7 Ø3 Ø3 Ø4
9Ø4Ø-  2B Ø4 7B Ø4 ØØ ØØ ØØ ØØ
9Ø48-  ØØ ØØ ØØ ØØ 25 27 2D 36
9Ø5Ø-  3E 3F 24 24 2C 2E AC 15
9Ø58-  2E 3E 2E 1E 1E 37 27 37
9Ø6Ø-  Ø7 38 38 2Ø 2C 64 36 3E
9Ø68-  2E 2E 2D 2D ØØ ØC 2E 2C
*
```

Notice that the first byte is the number of shapes (hex) in the table. The second byte is unused, since it would have been the high-byte of the "# of shapes in the table" but only 255 ($FF) are allowed, so it's nothing at all. The next byte is the low byte of the distance from the first byte of the index to the first byte of the first shape in the table. The next byte is the high byte of the distance from the first byte of the index to the first byte of the first shape in the table. After that is low and high distance to shape #2, then #3, and so on.

This means that in a regular (the table shown is NOT regular) shape table the first shape will be at the shape table address plus 2+(2X# of shapes in table), unless extra room at the end of the index has been allocated for addresses of future shapes.

See the programming manual (Applesoft) for more details. The way vectors and plotting works is explained fairly well. You can move or plot or do both in a shape byte. This means you can draw part of a shape and then move to a new location and draw some more. A shape can therefore be many shapes or contain many diverse elements. It must all be the same color though, if it's a vector (or hplot) shape. Block shapes may contain many different colors.

The best way we know of to actually draw vector shapes is with our Super Shape Draw and Animate Package. Or do a few the "hard" way via the Applesoft Program Manual. We've included some vector shape tables in this system, by the way. They are on 28D, mostly.

There are several vector shape programs on 28A, such as E,G,H,I, and parts of 3 and 4. Also, 5 of 28C is a special program that actually converts either hplot-shapes or block-shapes into vector shapes.

A few things to remember about vector shape drawing and usage:

(1) Use correct pokes at 232 and 233 to tell the computer where the shape table is located. An example would be POKE 232, Ø: POKE 233, 144. This would be for a table that starts at $9ØØØ (36864 dec.).

(2) If you forget a length or address, then BLOAD the table and ?PEEK (43634)+ PEEK (43635) * 256 for the address and ?PEEK (43616) + PEEK (43617) * 256 for the length.

(3) If you forget the # of shapes in a table then ?PEEK (addr.).

(4) Remember that HIMEM: 36864 will protect your table at $9ØØØ --- failure to protect it will lead to bad shapes or bombed programs or errors.

(5) After every HGR or HGR2 will come ROT, SCALE, and HCOLOR.

(6) If you want mixed-screen with HGR2, POKE-163Ø1, Ø.

(7) If you want full-screen with HGR, POKE-163Ø2, Ø. (Mixed gives 4 lines of text, full gives none.)

(8) If you want to jump to all text without erasing POKE-163Ø3, Ø: POKE-16298, Ø.

(9) If you want to jump back to graphics without erasing POKE-163Ø4, Ø: POKE-16297, Ø.

(Points 6-9 apply to all graphics programs)

(10) Use vector shapes for shapes to be rotated, enlarged, or shrunk. (If you need your block-shapes to be different sizes/rotations, use 5 of 28C and turn them into vector shapes.)

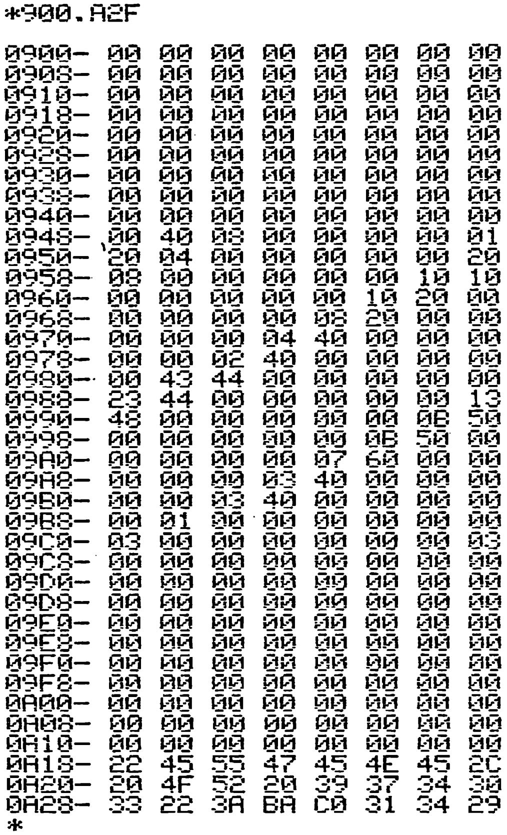
8B BLOCK-SHAPES

Here is a block shape:

(PRINTOUT #4)

Here is the shape table for this shape:

(PRINTOUT #5)



```
*900.A2F

0900- 00 00 00 00 00 00 00 00
0908- 00 00 00 00 00 00 00 00
0910- 00 00 00 00 00 00 00 00
0918- 00 00 00 00 00 00 00 00
0920- 00 00 00 00 00 00 00 00
0928- 00 00 00 00 00 00 00 00
0930- 00 00 00 00 00 00 00 00
0938- 00 00 00 00 00 00 00 00
0940- 00 00 00 00 00 00 00 00
0948- 00 40 03 00 00 00 00 01
0950- 20 04 00 00 00 00 00 20
0958- 08 00 00 00 00 00 10 10
0960- 00 00 00 00 00 10 20 00
0968- 00 00 00 00 08 20 00 00
0970- 00 00 00 04 40 00 00 00
0978- 00 00 02 40 00 00 00 00
0980- 00 43 44 00 00 00 00 00
0988- 23 44 00 00 00 00 00 13
0990- 48 00 00 00 00 00 0B 50
0998- 00 00 00 00 00 0B 50 00
09A0- 00 00 00 00 07 60 00 00
09A8- 00 00 00 03 40 00 00 00
09B0- 00 00 03 40 00 00 00 00
09B8- 00 01 00 00 00 00 00 00
09C0- 03 00 00 00 00 00 00 03
09C8- 00 00 00 00 00 00 00 00
09D0- 00 00 00 00 00 00 00 00
09D8- 00 00 00 00 00 00 00 00
09E0- 00 00 00 00 00 00 00 00
09E8- 00 00 00 00 00 00 00 00
09F0- 00 00 00 00 00 00 00 00
09F8- 00 00 00 00 00 00 00 00
0A00- 00 00 00 00 00 00 00 00
0A08- 00 00 00 00 00 00 00 00
0A10- 00 00 00 00 00 00 00 00
0A18- 22 45 55 47 45 4E 45 2C
0A20- 20 4F 52 20 39 37 34 30
0A28- 33 22 3A BA C0 31 34 29
*
```

Notice that no bytes are special and that you can't tell how many shapes there are or anything else about the shape from examining memory. But if you stare at the memory bytes for a bit, you'll begin to be able to tell that the actual shape width must be 7 so it'd get called 6. Another thing you can see is that way too much extra space around the shape was included in the block of memory included in this shape. This shape could have been 2X21, but the block that was scanned for this shape was called 6X39, which means actual dimensions of 7X40.

The actual reason the shape was saved in such an extravagant fashion is that it is a test shape built to be able to handle any amounts of logical shifting in any direction. More on that later.

The way you tell that a lot of bytes were wasted is to notice all the zeroes. This shape, whose **actual** dimensions are 7X40, takes up 280 bytes, which is 7 times 40. The shape **in** the block-shape doesn't require 7X40, but the actual block-shape itself is a data array of 7X40. This would be very similar to a text file block-shape data array like NUM (7, 40) which is a **2-dimensional array.**

Let's notice that we've said that each shape is given 256 bytes (1 page) of memory. Then let's notice that this shape, called MANDG on 28B, is 280 bytes long. What gives? Simple: this shape is shape #1, but is taking up part of shape #2's addresses, so the next shape, if there ever is a next shape, must be shape #3 (in truth there is only 1 shape in this table). There can be NO shape #2. Notice that the bottom 3 lines in the list-out which are not part of the shape, are 281-304 bytes away from $900. You can tell at a glance where this shape ceases. Often it's not this easy, but this time it is.

But what are the implications of being able to notice that a shape is (actual dimensions) 7 wide by 40 high? Simple, it means that if you forgot the shapes' dimensions, or never knew them, or are too lazy to look them up in your notebook, you can tell them by examining the memory after BLOADing the shape table. (Actually, in a one-shape-long table, ?PEEK (43616) + PEEK (43617) * 256 would be the easiest way to get the length.)

Perhaps you got lost when "we" determined the shape was 7 wide. Stare at the list-out. You'll notice that when the shape's non-zero bytes appear, they seem to be doing so every 7 bytes, and the man is usually 2 bytes (14 dots) wide. This gives **actual** width, which you'll **call** 6, for reasons cited in the last chapter.

Now, when you notice that there is no other shape after #1 (when you ask to see #2 when using program 3 of 28A, you get garbage) and that the memory has a dramatic change at $A18, you should be able to reason out that shape #1 ends there. To get the length of the shape without using PEEKS, you simply add page 9 ($900-$9FF) (256 bytes or $100) to the 3 rows of 8 bytes used on page A. (A comes after 9 in hex.) So 256+ (3X8)=280. Now divide 280 by 7 and you get 40 as **real** height.

Go ahead and use 3 on 28A to inspect MANDG on 28B. Try VT=0, VB=39, HR=6, HL=0 and then try VT=50, VB=89, HR=27, HL=21. The shape is fine. Notice that the width added to HL should always give HR, and the height added to VT should always give VB. This is as it should be. Also notice that I'm talking about the height you say, and the width you say, not the actual width and height, which one gets by adding one to either dimension. Keep in mind that all the programs and subroutines already automatically add one to the height and width dimensions, in effect. So don't YOU ever add one **also**, by giving **actual** height and width in inputs. Okay?

Again, VB−VT=height and HR−HL=width. If the numbers you give don't jive with these equations, give better numbers until they **do** jive.

When you actually create a block-shape you'll be told the dimensions to jot down. Since you'll never be told "actual" dimensions, but always the "actual−1" dimensions that I've been convincing you to utilize throughout these chapters, no confusion will arise. The **only** time you'll need to remember "add one for **actual**" is when you examine memory.

Where will you **draw** your shapes? In program 8 of 28B you can draw and save hplot-shapes. In program 2 of 28A you can draw and save block-shapes. In no program are vector shapes drawn, but we have a real nice Super Shape Draw and Animate Package for that purpose. (Also, you may draw vector shapes by hand via the Applesoft Program Manual, or use the dozens of shapes supplied with this system.) You may make all the vector shapes you want by merely drawing shapes in 2 of 28A, saving them as block-shapes in 4 of 28A, and then converting them to vector shapes in 5 of 28C (you'll be using your own initialized disks to store shapes on --- the program disks are too full).

The easiest way to make a block-shape is to scan. If you saw the movie about Scanners this may be bad news. Well, cheer up. Not **that** type of sci-fi nonsense. What scanning means is that you mark off a certain section of the hi-res screen that has something that you'd like to save as a block-shape, and then you save it via a scanning process. Here's how it works:

You could start by doing one or more of the following: drawing pictures with our Instant Graphics (Block Shapes) program (2 on 28A) and/or using the special shape-drawing option by hitting J and using keyboard commands, or load in one or more already-saved block-shapes into 4 of 28A (our scanner program), or load in 1 or more vector shapes or a combination of block and vector shapes.

No matter where you are, or what program you're in, you'll go to either 4 of 28A or 6 of 28D for general block-shape scanning and saving. There's an option at the beginning of this program that lets you enter it without erasing the screen. This will turn out to be quite useful.

I recommend doing really far-out vector shapes with Super Shape Draw and Animate and scanning them into block-shapes via 4 of 28A. But 2 of 28A would be second choice, it does plenty that most shape programs can't.

Okay so what is **scanning**? Simple. Define a part of the screen, with the use of game paddles, while in 4 of 28A, and then save the bytes as raw data in a binary file, called a block-shape table. Each table holds up to 5888 bytes or 23 shapes.

If you know anything about the way the hi-res screen is mapped out you'll know that the addresses on the hi-res screen are mixed up badly. So how do we scan a mixed-up mess of addresses in a rational, effective way? The answer --- the same way porcupines make love: very carefully. **And** by the use of HPOSN. This routine is found in Applesoft at $F411. By putting hor. low into the X register, hor. high into the Y register, and vert. into the accumulator and then doing a JSR $F411, which means "jumping over to that subroutine and returning when done" ("done" is signalled by RTS) --- we end up with the address of any byte in either hi-res page. This subroutine is the **unscrambler** of hi-res addresses. The unscrambled addresses are put into page Ø.

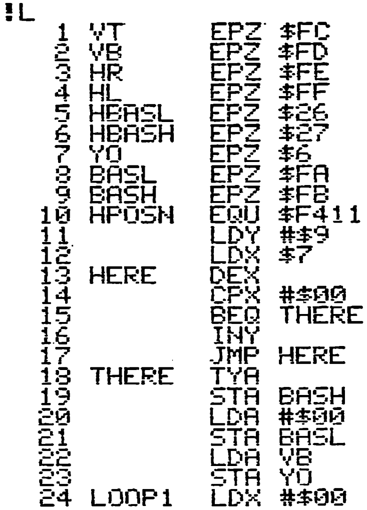
HPOSN also does other things, such as deal with color tables, bit tables and internal/external cursor equivalence. It's not important to define these actions further at this time, but one thing is important: routines that scan and save or draw and animate block-shapes don't need any part of HPOSN except the unscrambling part (that puts hi-res page low-byte address into $26 and hi-res page high-byte address into $27 for later indexed indirect addressing use).

So the remainder of HPOSN is a waste of time for block-shapes. That's why YTABLE (my term) was invented (not by me) a couple of years ago. What this table is is a way of getting your hi-res address quicker than HPOSN by use of tables rather than calculations such as those done by HPOSN. There's a disadvantage, though. If you use one hi-res page the tables take up 1 1/2 pages of memory, and if you use 2 hi-res pages, 2 1/4 pages get taken. If you've got the room, you'll get at least a 20% speed increase. If not, c'est la vie. Disk storage space is a factor here too, as is availability of zero-page addresses not being used for anything.

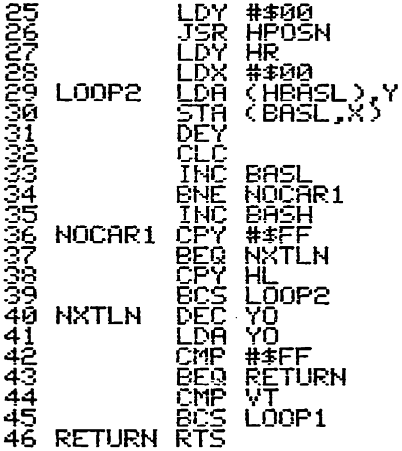
Anyway, for a quick idea of what these tables are about, see page 21 of the new white Apple Reference manual. The list of addresses from $2000 to $2320 are some of the addresses in the table. But let's go on now --- there's a chapter on YTABLES later on.

So a scanner program is one that reads all the data in a previously defined block of memory and puts this data into shape tables. In the case of the programs on this disk, shape #1 bytes get put into $900, shape #2 goes into $A00, and so on until shape #23 goes into $1F00. If you were using only hi-res page 2 there'd be room for lots more shapes from $2000 to $3F00 --- 32 more to be exact. But who needs that many shapes anyway? Not I, said the little red hen. Here's a scanner program list-out:

PRINTOUT #11



```
!L
   1 VT      EPZ $FC
   2 VB      EPZ $FD
   3 HR      EPZ $FE
   4 HL      EPZ $FF
   5 HBASL   EPZ $26
   6 HBASH   EPZ $27
   7 YO      EPZ $6
   8 BASL    EPZ $FA
   9 BASH    EPZ $FB
  10 HPOSN   EQU $F411
  11         LDY #$9
  12         LDX $7
  13 HERE    DEX
  14         CPX #$00
  15         BEQ THERE
  16         INY
  17         JMP HERE
  18 THERE   TYA
  19         STA BASH
  20         LDA #$00
  21         STA BASL
  22         LDA VB
  23         STA YO
  24 LOOP1   LDX #$00
```

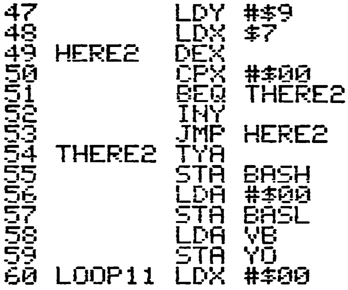


```
                LDY  #$00
                JSR  HPOSN
                LDY  HR
                LDX  #$00
        LOOP2   LDA  (HBASL),Y
                STA  (BASL,X)
                DEY
                CLC
                INC  BASL
                BNE  NOCAR1
                INC  BASH
        NOCAR1  CPY  #$FF
                BEQ  NXTLN
                CPY  HL
                BCS  LOOP2
        NXTLN   DEC  YO
                LDA  YO
                CMP  #$FF
                BEQ  RETURN
                CMP  VT
                BCS  LOOP1
        RETURN  RTS
```

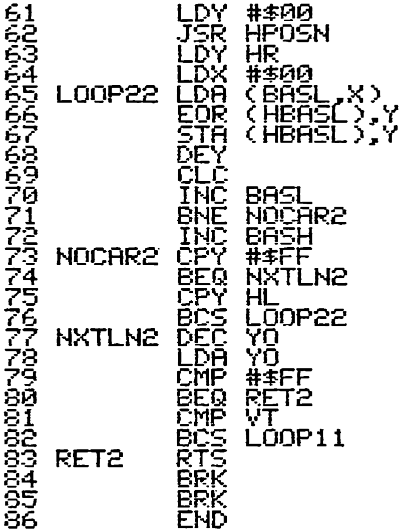
As you can see, the routine starts out by defining which page you'll be storing the data on, according to which shape # it finds in $7, which was previously entered, before this subroutine was called, by POKE 7, shape #.

Next it takes the VB and finds the address of the coordinates 0, VB (X=0)(Y=VB). Then it grabs all the data from the line (hor.) it's on that is included from HL to HR on that line and stores these bytes in the shape table, and then moves up a line and does it again. Once VT is reached it lets that be the last data-grab. It's fairly simple. There was a Call Apple article about some of this not too long ago. Unfortunately it was for Integer and has zero-page variables that would be no good in Applesoft. It also had no shape # reconciler at the start of the routine. It also had no EOR therefore it wouldn't XDRAW or erase. Anyway I tried to use the same variable names and such to make it easy for all of you already familiar with this type of routine. Actually, **getting** block-shapes **into** shape tables is a piece of cake, it's what you do once you begin drawing and animating these shapes that takes the True Grit. So here's what it takes to get the shape data from a table and place in onto the hi-res page:

PRINTOUT #12



```
                LDY  #$9
                LDX  $7
        HERE2   DEX
                CPX  #$00
                BEQ  THERE2
                INY
                JMP  HERE2
        THERE2  TYA
                STA  BASH
                LDA  #$00
                STA  BASL
                LDA  VB
                STA  YO
        LOOP11  LDX  #$00
```



```
61         LDY #$00
62         JSR HPOSN
63         LDY HR
64         LDX #$00
65 LOOP22  LDA (BASL,X)
66         EOR (HBASL),Y
67         STA (HBASL),Y
68         DEY
69         CLC
70         INC BASL
71         BNE NOCAR2
72         INC BASH
73 NOCAR2  CPY #$FF
74         BEQ NXTLN2
75         CPY HL
76         BCS LOOP22
77 NXTLN2  DEC YO
78         LDA YO
79         CMP #$FF
80         BEQ RET2
81         CMP VT
82         BCS LOOP11
83 RET2    RTS
84         BRK
85         BRK
86         END
```

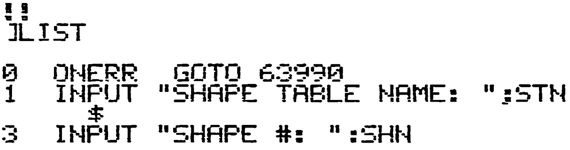
Remarkably similar to the scanning program, this little routine has the XDRAW characteristic just because of the EOR (HBASL), Y line. Leave that line out and you have a block-shape that DRAWS rather than XDRAWing, which makes it no longer erase.

These scanning and drawing routines are enough for you to make a start at block-shapes, but in order to really have block-shapes work better than vector shapes for you, you'll need quite a few routines that facilitate efficient and easy-to-understand block-shape manipulation and animation.

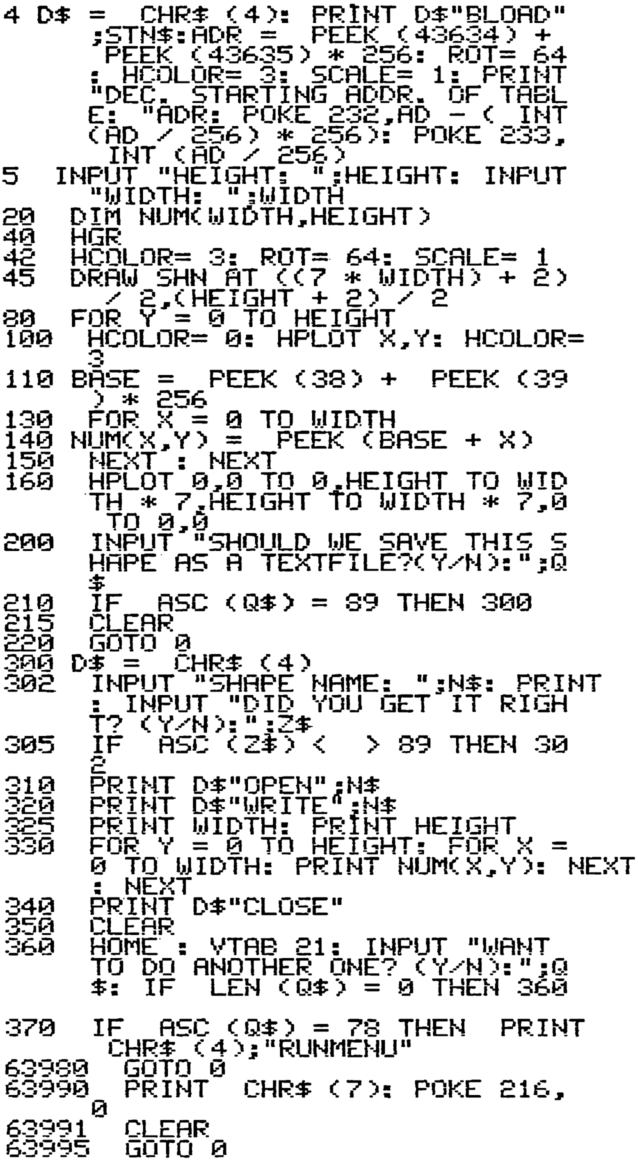
8B1 TEXT FILE BLOCK-SHAPES

On 28A there's program 7 to scan block-shapes and save them as text files. 8 on 28A views these shapes, and 9 on 28A views them upside down. Also on 28A is a text file QQ, which can be used by #8 or #9. If you use #7 to make a text file, don't save it on 28A --- use a data disk. So once you've given the shape's name, switch disks before you hit RETURN. And switch back to 28A if you say you don't "want another one". Here is program #7:

PRINTOUT #8



```
]LIST

0  ONERR  GOTO 63990
1  INPUT "SHAPE TABLE NAME: ";STN
    $
3  INPUT "SHAPE #: ";SHN
```



```
4 D$ =  CHR$ (4): PRINT D$"BLOAD"
     ;STN$:ADR =  PEEK (43634) +
      PEEK (43635) * 256: ROT= 64
     : HCOLOR= 3: SCALE= 1: PRINT
     "DEC. STARTING ADDR. OF TABL
     E: "ADR: POKE 232,AD - ( INT
     (AD / 256) * 256): POKE 233,
      INT (AD / 256)
5  INPUT "HEIGHT: ";HEIGHT: INPUT
     "WIDTH: ";WIDTH
20  DIM NUM(WIDTH,HEIGHT)
40  HGR
42  HCOLOR= 3: ROT= 64: SCALE= 1
45  DRAW SHN AT ((7 * WIDTH) + 2)
      / 2,(HEIGHT + 2) / 2
80  FOR Y = 0 TO HEIGHT
100  HCOLOR= 0: HPLOT X,Y: HCOLOR=
     3
110 BASE =  PEEK (38) +  PEEK (39
     ) * 256
130  FOR X = 0 TO WIDTH
140 NUM(X,Y) =  PEEK (BASE + X)
150  NEXT : NEXT
160  HPLOT 0,0 TO 0,HEIGHT TO WID
     TH * 7,HEIGHT TO WIDTH * 7,0
      TO 0,0
200  INPUT "SHOULD WE SAVE THIS S
     HAPE AS A TEXTFILE?(Y/N):";Q
     $
210  IF  ASC (Q$) = 89 THEN 300
215  CLEAR
220  GOTO 0
300 D$ =  CHR$ (4)
302  INPUT "SHAPE NAME: ";N$: PRINT
     : INPUT "DID YOU GET IT RIGH
     T? (Y/N):";Z$
305  IF  ASC (Z$) <  > 89 THEN 30
     2
310  PRINT D$"OPEN";N$
320  PRINT D$"WRITE";N$
325  PRINT WIDTH: PRINT HEIGHT
330  FOR Y = 0 TO HEIGHT: FOR X =
     0 TO WIDTH: PRINT NUM(X,Y): NEXT
     : NEXT
340  PRINT D$"CLOSE"
350  CLEAR
360  HOME : VTAB 21: INPUT "WANT
     TO DO ANOTHER ONE? (Y/N):";Q
     $: IF  LEN (Q$) = 0 THEN 360

370  IF  ASC (Q$) = 78 THEN  PRINT
      CHR$ (4);"RUNMENU"
63980  GOTO 0
63990  PRINT  CHR$ (7): POKE 216,
     0
63991  CLEAR
63995  GOTO 0
```

Here's what will happen in this program: You'll give a vector shape table name. Then you'll be asked for the height and width. "How many dots tall is it?" is the essence of the height question. But the width question wants your answer in bytes, not bits. Remember, the screen is only 40 bytes wide. Leave plenty of extra room, especially if your shape's reference point is off-center, in which case you'd be wise to over-guess dimensions 3 times over.

Line 45 draws in the shape and line 80 begins the scanner routine. **Line 100** plots an invisible dot for the sole purpose of getting HPOSN into action, since HPOSN is done as the first part of an HPLOT. The result of HPOSN is to get the low and high bytes of the current hi-res "cursor's" address into $26 and $27. Since $26 is 38 (dec.) and $27 is 39 (dec.) then this should shed light on line 100. (If you're still hazy about hex and dec just remember to take a number like $26 and multiply the 2 by 16 and then add the 6. (2X16)+6=38.)

Line 110 calculates the base address of the hi-res cursor. Remember, the high byte is "how many 256's" (pages) in the address, so to be able to add it to the low byte we must first get the actual number it represents by multiplying it by 256. Think of it like this: the high byte tells which page ($20 through $3F or $40 through $5F) on the hi-res screen (1 or 2) we're on, while the low byte tells how far (out of a possible 255, since at 256 we're on the following page) into that page we are.

If line 110 found out that $26 held $80 and $27 held $21 then $2180 would be the hi-res cursor's current address. Of course, this is merely the address of the left-most address on the screen. You'll need rightward displacements greater than byte 0 on many if not most of the bytes you save. (Incidentally, this program as well as the 2 that follow it start shape blocks in the upper left at byte 0 and horizontal line #0. It makes for fewer inputs.)

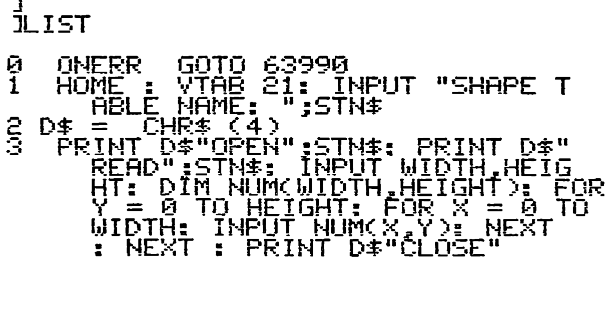
Line 130 starts a FOR-NEXT that takes care of learning the data for each byte on the line you're on, **and line 140** stores these bytes in a 2-dimensional array.

Line 150 makes sure all bytes of all lines in the specified block get included, **line 160** draws a rectangle around the chosen block, and **line 200** lets you decide if you want to save the block-shape as defined.

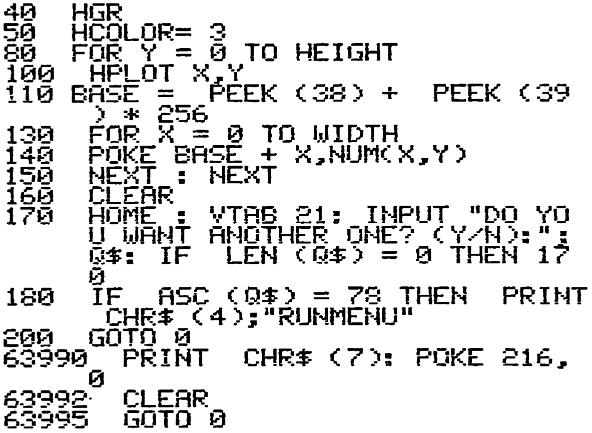
The remainder of the program saves the block-shape in a text file.

This is simple scanning and saving. Now let's look at pulling the information out of the text file and redrawing the shape. This is program #8:

PRINTOUT #8



```
]
]LIST

0  ONERR  GOTO 63990
1  HOME : VTAB 21: INPUT "SHAPE T
     ABLE NAME: ";STN$
2 D$ =  CHR$ (4)
3  PRINT D$"OPEN";STN$: PRINT D$"
     READ";STN$: INPUT WIDTH,HEIG
     HT: DIM NUM(WIDTH,HEIGHT): FOR
     Y = 0 TO HEIGHT: FOR X = 0 TO
     WIDTH: INPUT NUM(X,Y): NEXT
     : NEXT : PRINT D$"CLOSE"
```

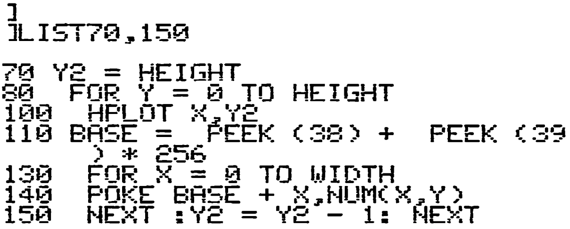


```
40  HGR
50  HCOLOR= 3
80  FOR Y = 0 TO HEIGHT
100  HPLOT X,Y
110 BASE =  PEEK (38) +  PEEK (39
   ) * 256
130  FOR X = 0 TO WIDTH
140  POKE BASE + X,NUM(X,Y)
150  NEXT : NEXT
160  CLEAR
170  HOME : VTAB 21: INPUT "DO YO
     U WANT ANOTHER ONE? (Y/N):";
     Q$: IF  LEN (Q$) = 0 THEN 17
     0
180  IF  ASC (Q$) = 78 THEN  PRINT
      CHR$ (4);"RUNMENU"
200  GOTO 0
63990  PRINT  CHR$ (7): POKE 216,
     0
63992  CLEAR
63995  GOTO 0
```

This program is the reverse of the last, and the first few lines merely read the array in the text file. The main difference between this program and the last is that in the last one line 140 dumped the bytes at each address PEEKED into, into a 2-dimensional array (NUM(X,Y)=PEEK(BASE+X)), while program #8 POKED the array bytes of the text file into the block-shape area on the hi-res page (POKE BASE+X, NUM(X,Y)).

The last text file program #9 does what #8 does but turns the shape upside down in the process:

PRINTOUT #10



```
]
]LIST70,150

70 Y2 = HEIGHT
80  FOR Y = 0 TO HEIGHT
100  HPLOT X,Y2
110 BASE =  PEEK (38) +  PEEK (39
   ) * 256
130  FOR X = 0 TO WIDTH
140  POKE BASE + X,NUM(X,Y)
150  NEXT :Y2 = Y2 - 1: NEXT
```

What this program does is to get the bytes for the highest vertical coordinates and put them at the lowest vertical coordinates of the shape and then work its way through the shape, effectively inverting it. This program might be quite useful in creating reversed or inverted duplicates of various vector shapes. Usual vector shape rotation has no way to invert shapes --- if you rotate 32 the shapes are upside down **and** reversed. If you only want upside down **or** reversed, rotation is not for you.

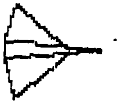
If you'd rewrite the scanning and text file saving program with the option to include rotational factor as an input, you could not only (by going to 4 of 28A afterwards) save non-reversed upside down versions of shapes, you could also put a rotational factor of 32 into the shape before scanning it, which would mean that if you redraw it with 9 of 28A it would be turned upside down **again** and end up right-side up but reversed from right to left.

To get either hplot or block-shapes into different rotations or sizes, use 5 of 28C to convert them into vector shapes.

8C HPLOT SHAPES

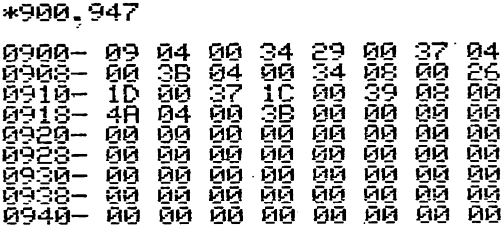
Here is an hplot-shape:

PRINTOUT #6



Here is a shape table for that shape:

PRINTOUT #7



```
*900.947

0900-  09  04  00  34  29  00  37  04
0908-  00  3B  04  00  34  08  00  26
0910-  1D  00  37  1C  00  39  08  00
0918-  4A  04  00  3B  00  00  00  00
0920-  00  00  00  00  00  00  00  00
0928-  00  00  00  00  00  00  00  00
0930-  00  00  00  00  00  00  00  00
0938-  00  00  00  00  00  00  00  00
0940-  00  00  00  00  00  00  00  00
```

Notice it has 9 points in the table but only 7 main points in the actual shape. The reason is simple: The point Hor. low=Ø4, Hor. high=ØØ, Vert.=34 is closed on once in addition to being a starting point, and the point Ø4, ØØ, 3B is closed on once, which means that both of these points get used twice. Where the wings meet near the front of the jet is 2 points, not one, which is why it looks like there are only 6 points at first. (Those 2 points are 1D, ØØ, 37 and 1C, ØØ,39.) It's easy to arithmetically figure out that 3 bytes per point will give a nine-point shape a length of 28, since the first byte is the point counter and 1+(3X9)=28. Byte 2 in a point must be Ø or 1, as it's a hor. high byte. Since 279 is the maximum coordinate and 256 is where the high-byte turns into one, you can see that that last byte doesn't do too much good for hplot demonstration packages. In actual programs that you create you may wish to add dealing with hor. high byte in your mach. language subroutines. (You'll get 9% more horizonal space.)

Since such little memory is used by this type of shape we might point out that by leaving out the reading in of the high byte (hor.) and leaving that high byte out of the shape table, **even less space would be needed.**

It seems a shame to waste so many bytes (256 each) on shapes that will probably vary from 10 to 37 in length. But, as we've said, there's nothing to stop anyone from using indexes for their hplot-shapes that find hundreds of shapes in a table. Or a person could add $40 (64 dec.) several times each page for each page of table space so that table capacity quadrupled.

An hplot-shape can be turned into a block-shape in 2 ways, although this may never be appropriate. One way is to view the shape with 7 of 28B and then put in 28A **before** you leave that program. This way when you leave the program and "MENU" is run, it will be the 28A MENU. This is, incidentally, the way one goes between system disks (28A and 28B) without rebooting. When you get to MENU hit 4 for our scanner program. Use the paddles to define the shape and then save it as a block-shape.

The second way to make a block-shape out of an hplot-shape is weird. Get into 2 of 28A and use the "paddle-draw" option (which requires the game paddles). Draw your hplot shape (while in the record mode) according to the instructions in the next chapter. When it's completed, save the shape as an Instant Graphics text file, remembering to switch to a data disk. Now do option #4, while still in this program (2 of 28A) and **see** the saved drawing you just saved. All the shakiness is now gone from your paddle-drawn hplot-shape. Now go to 4 of 28A and save this shape as a block-shape. There's a lot of memory waste in either of the above 2 methods --- why not use hplot-shapes as hplot-shapes and block-shapes as block-shapes instead?

Hplot shapes, as already noted, are easy to use, take up the least amount of bytes, are quick, and efficient, may be done in any color regardless of original color, and are best for shapes composed of a series of straight lines.

An hplot (or block)-shape may be turned into a vector shape in 5 of 28C.

8D INSTANT GRAPHICS (BLOCK-SHAPES)

This is going to be a long one. (Is it my fault I decided to make this program do a lot?) So take a deep breath and stretch a bit, and perhaps say the magic words: ret up moc elppa. (I'll let you work on that a bit. The first person to call us --- and only the first --- with the answer to the above can buy any other piece of software we sell for half price!)

This program will let you draw, quickly and easily, at the touch of a key or 2. You can draw squares, 2 types of rectangles, circles, ellipses, slide-walls from 8 directions, fireworks, dots, snow, triangles, random lines, horizontal or vertical or diagonal lines, frames, dynamic enclosures, backgrounds, and so on. You may fill shapes with any of the hi-res colors and stop the filling at any point and go on. You may create bushes, trees, houses, etc. You may change colors at any time. A shape may be filled with many different bands (or donut shapes) of color. You may monitor your coordinates. There will be a floating dot-cursor for orientation. You may see command options at any time by hitting space bar.

You may save drawings and have them retrieved and drawn back again on the screen, step by step, as if the magic. You may use either **regressive** or **symmetry** mode; in the former most shapes will get smaller and smaller as you draw them further to the right, which is great to convey depth and perspective; in the latter, shapes end up the same size regardless of hi-res positioning. I'm referring to shapes that are (for example) of M (medium) size will be that precise size no matter where they are, if you're in symmetry mode, while in regressive mode M-sized shapes are proportioned to their distance from the right edge of the screen. There are 8 sizes to choose from, and the regressive mode expands on the number of possible sizes enormously.

You may choose to have drawing sounds accompany your drawing, and also accompany your retrieved drawing "play-backs".

You may paddle-draw, which will result in shaky drawings, and then save the result --- but watch what happens when you retrieve the drawing: the drawing is perfectly smooth and flawless!

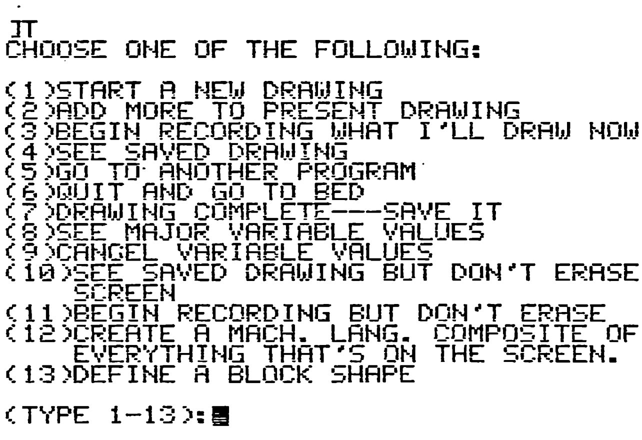
You may monitor the major variables you're using, and also monitor the precise screen coordinates you're at.

You may create 34-sector machine language composite drawings of from 1 to any amount of previously-saved drawings.

You may stop at any time and go to the scanning program to save what you've done as a block-shape --- this allows saving of any or all parts of the screen, regardless of size.

Here is the Instant Graphics (Block-Shapes) menu:

PRINTOUT #13

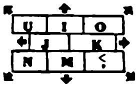


```
]T
CHOOSE ONE OF THE FOLLOWING:

(1)START A NEW DRAWING
(2)ADD MORE TO PRESENT DRAWING
(3)BEGIN RECORDING WHAT I'LL DRAW NOW
(4)SEE SAVED DRAWING
(5)GO TO ANOTHER PROGRAM
(6)QUIT AND GO TO BED
(7)DRAWING COMPLETE---SAVE IT
(8)SEE MAJOR VARIABLE VALUES
(9)CANCEL VARIABLE VALUES
(10)SEE SAVED DRAWING BUT DON'T ERASE
    SCREEN
(11)BEGIN RECORDING BUT DON'T ERASE
(12)CREATE A MACH. LANG. COMPOSITE OF
    EVERYTHING THAT'S ON THE SCREEN.
(13)DEFINE A BLOCK SHAPE

(TYPE 1-13):█
```

First let's look at command option #13, **Define A Block-Shape.** What this command does is send you to 4 of 28A, the scanner/saver program. No matter what types of things (or how many) you've gotten onto the hi-res screen, you may save some or all of them in block-shapes of whatever size you like. (Remember that if your table is over 5888 bytes long you'll be crashing into the hi-res page that starts at $2000.)

Now let's look at the cardboard drawing card. It's also in the appendix at the back of this manual. Check out command J. It allows you to draw in a way similar to the way the drawing programs in our Super Shape Draw and Animate Package draw.

Point-plot movement commands:



Only two commands:
P=plot on/off Q=quit

The above commands are for subroutine J only. They allow you to draw delicate little shapes quickly and easily. Larger ones are fast too. You may put transparent (cellophane?) material over any picture and trace it, and then tape it to your monitor screen and again trace this picture ---this time you'll trace the tracing with the plot/move keyboard commands.

Now the J subroutine utilizes an XDRAWN dot. Since XDRAWING means draw in the color opposite what's there now, this means that if you draw a line of white dots but then back up and go over these dots again, they'll be gone, right? Well, almost. But since the U-TURN dot changed to its opposite only once, it alone gets left visible. The rest are gone quicker than program secrets at a software pirate's convention.

Now run the program (2 on 28A) and try it out:

1) When asked if you want sound, hit Y (and RETURN).

2) You'll hear 2 buzzes --- these mean "ready to draw".

3) Hit J and watch the blinking dot-cursor quit blinking.

4) You're now in keyboard command **plot-mode**; if you were to hit P you'd be in **move-only mode.**

5) Hit 5 I's and then 5 M's; see what we mean about U-turns? Try it again: Hit Q to Quit, then SPACE BAR for command options, then 1 and RETURN for Start Over, then J for some more keyboard plots.

6) Hit the 5 I's again but this time erase better on that U-turn: hit P, I, and P and then a couple M's.

7) No littering on this U-turn! You're erasing old plots without a trace. But what if you decide to go upwards again? Hit a couple I's again. Just like with the first U-turn, the turning dot messed up.

8) Use Q, SPACE, 1, RETURN and J again. Then hit 5 I's and then P, I, P, and then 3 M's. Now try P, M, P. A successful U-turn again. The moral seems to be if you wish to do U-turns while erasing or redrawing, use the P command, the last move command used, and then the P command again before beginning to plot in the opposite direction. It keeps that odd turnaround dot from actually being plotted a different amount of times from the rest of the dots.

9) Let's try direction change as erase turns back into draw. Hit Q, SPACE, 1, RETURN, J, I, I, I, I, I, N, N, N, N, N, N, N, N.

10) Let's say you need a tiny triangle and the line you're on is too long. Hit P, N, P, O, O, O until the bottom dots of the 2 sides of your triangle are level.

11) Hit P, O, P, and then L, L, L, L. The triangle's finished, right?

12) Wrong! We fixed it so block-shape drawing would be quite like vector shape drawing (with Super Shape Draw!). You have to throw in one extra dot in the same direction you were last traveling. If you don't, there'll be a dot missing at the end of your shape. But don't take my

word for it --- hit Q to quit, since the shape looks perfect. Now move a game paddle a bit. See?

13) You're missing a dot. You're shape is like a Flash Gordon serial ---no ending. Here's a cheap and dirty fix: move the dot-cursor onto the missing dot and hit D. It's all fixed.

14) But go through the creation of the triangle one more time, complete with "errors". But this time once the figure is "closed" and all looks perfect, remember the golden rule:

He who forgets the extra dot
Will surely someday go to pot.

15) Move that one last dot rightwards, even though it seems to erase the last dot. Then hit Q to Quit and begin moving paddles again. Voila ---success!

If that's the only block-shape you needed your could just hit SPACE for command options and then 13 to get to the scanner and define/save it. But there's nothing to stop you from trying to fill the screen full of groovy shapes or figures before going off to Scansville. Go for it! Try all the commands, from A to Z. You'll need a few extra instructions on some figures. But if you study the drawing card (Appendix) carefully, most things will be a snap.

If you hit G, X, N, Y, H, U, K, V you'll get slide-walls of the last specified color. Stop these slide-walls with paddle button 0, which is why the 2 asterisks on these figures refer you to a comment that points at button #0 as the stopper.

If you want an unfilled circle try 0 and Y and get a very short one (Y=very short, in column 3). The drawing of the 6 figures after the slide-walls can be done with or without filling, and the filling can be stopped at anytime with the paddle 0 button. In fact, you needn't even draw the figure before you fill it. Move the paddles and say you want an ellipse (I) that is filled (F) and short (E). Hit paddle 0 button before it's done. Now change colors by Color (C) and then Violet (V). Fill the ellipse again but hit the button even sooner. Starting to see what all these commands are for? Hit C for color, B for black, O for circle, F for fill, and O for shortest. The amount and types of tricks this program has up its sleeve is endless, and the program is **faster** without choosing sounds at the beginning.

If you'd like an 88-page manual most of which is about this program, ask us, it's available.

Other subroutines are available that stop when you hit paddle 0 button, such as S for snow, LR for random lines, and E for enclose. Don't forget, color is still set to black --- change it to orange or something. If you hit B by mistake after the C and O (orange), then the entire scene will turn orange --- as B means **background.**

Try out P, F, Q, R, D, A to abort, W, T, and M. When something refuses to draw you're too close to an edge. The program will continue.

To try out the second column commands, you must first hit L for line. To use **guess** (G) you'd need a sequence like LGDE. The line's starting point will be random. For a rainstorm use LGDE and LGDS many times each.

For a diagonal line try LDM (example), which means line, diagonal, medium length. (Lines are calculated as fractions (varying from 1/1 to

1/128) of the way to either the right or bottom edge of the screen.) Try a short vertical line: L,V,S.

Command option #8 and #9 turn on and off major variable value monitoring.

A$ = draw what
P$ = what type of line
D$ = what size line or figure
J = X coordinate of ellipse center
K = Y coordinate of ellipse or circle center
H = X coordinate of circle center
D1$ = F if "fill this shape"
X = PDL X-axis coordinate
Y = PDL Y-axis coordinate
S1 = PDL button was pushed at this variable's # value to stop first 3rd of slide-wall filling action or to stop "snow" or "enclose" or "random lines" or PDL-drawing or figure filling
S2 = second 1/3 of slide-wall had a stop at this #
S3 = third 1/3 of slide-wall had a stop OR symmetry mode is in effect
C$ = HCOLORS (B or L=black. W or H=white. V=violet, G=green, U=blue, O=orange) default=W.

The order in which these variables will be seen on the screen is: A$, P$, D$, J, K, H, D1$, X, Y, S1, S2, S3, C$. Strings get skipped if they'll null (nothing). Other variables read as Ø if they're "nothing".

Now onto paddle-draw. The idea here is to hit 3 to get into **record-mode** from command options; then draw with L for line and P for PDL, after which you simply hit PDL#1 button, move paddles, hit PDL#1 button, move paddles, etc. until you're done, at which point you quit by hitting, as usual, PDL#Ø. The routine draws perfectly straight lines between each point the cursor was at when button #1 was pushed (over 32 lines in a row are too many to save --- the limit of steps saved into a text file is 32; you can check the step you're on with command M, but not during PDL-draw):

Now you're scratching your head because the routines lines are shaky, not straight. Well, what you see **now** is shaky, but the thing that got saved was point coordinates; so when the figure gets retrieved (#4 in command options) and viewed later, the shape **will** be straight.

This should remind you that we've already mentioned that you may draw HPLOT type shapes as block-shapes with PDL-drawing. These are hplot shapes, and with Instant Graphics text file Drawings this program can save (command #3 and #7) them so they come back as straighter text file hplot shapes, but going to 4 of 28A will turn anything into a **block-shape**, via scanning and saving.

To erase with Instant Graphics (Block-Shapes) change the color of part to be erased to background color by redoing it in background color, usually black, or perhaps you'll need to draw a figure bigger than the first shape or line to make sure your "black-outs" don't miss. Black-outs are also something you can do with slidewalls to **mold** one or more large, probably-central figures.

Draw a bunch of RFE after hitting RETURN for Regressive Mode, and then hit ESC for Symmetry Mode and do more RFE. Figure it out?

Remember, anything (however accidental) that gets created on the hi-res screen is fair game to save as a block-shape in 4 of 28A. This includes the fireworks (P) in this Instant Graphics (Block-Shapes) program, the keyboard-plotted J-shapes, the striped and orbited ellipses, etc.

Try lots of different coordinates with E, and stop it or try letting it go. X=217, Y=92 and X=147, Y=185 and X=89, Y=159 and X=46, Y=19Ø are all good combinations. Try them in black on a white background.

8E SUPER SHAPE DRAW

This package has no Super Shape Draw or other vector shape drawing programs. If you desire such a package, contact us.

There are many (over 2Ø) vector shape tables in this graphics package, however, such as Animals, Boom, Man, Spaceships, Media and Char. (found on 28D). To view any of these shapes use 3 of 28A ---there's an option to flip through the entire table. (Or there are several 28D programs that do it.)

There are several programs which involve vector shapes: the sample game on 28C uses BOOM (explosion simulator) shapes. The boot program on each disk uses CHAR. Superfont and RETRIEVE use CHAR also. Superfont is B of 28B and RETRIEVE is C of 28B. E of 28A uses a vector shape in MAN.

Vector shape utilities include 3 of 28A (partly), G of 28A, H of 28A, I of 28A. These are animation demos using 1 or 2-page flipping graphics and a byte/address/placement examination program that really dissects those vector shapes.

5 of 28C will convert block or hplot shapes into vector shapes.

A shapedraw program reads the keyboard to find which key you're hitting; the key you hit shows which of the 8 possible horizontal/vertical/diagonal directions you wish to plot or move in. Once the program knows the direction, it goes to routines that notice whether or not the plot flag is on or off (1 or Ø) and assign dec. #S to the vector that's been chosen. This must happen for each of the 3 sections that vector bytes are divided into:

Section:	$\overbrace{\text{C}}$	$\overbrace{\text{B}}$	$\overbrace{\text{A}}$	
Bit #:	7 6	5 4 3	2 1 Ø	These 8 bits=one shape byte
Meaning:	D D	P D D	P D D	(P=plot bit; D= direction bit

Column C must be downward, left, or right vectors only and must be **non-plotting** vectors, since column C has only enough room for directional bits, but not plot bits. If either plot bit is a 1 (plot bits are #5 and #2) then that means a dot gets plotted. Obviously the maximum # of plots per shape-byte is 2, since there are only 2 plot-bits per byte. When diagonal plotting happens, each byte is a move and a move-plot vector, so only one plot happens. It's the horizontal and vertical plotting that allows 2 plots per byte.

Once the proper #s are figured out of the 3 columns in a shape byte, the following line occurs:

```
B=C(1)+C(2)*8+C(3)*64: POKE LOC,B
```

This adds up the numbers in the 3 columns to get the final vector shape byte for that address. This number is POKED into the current address of the shape byte of the shape being drawn.

Actually, all of the above was the simple part of the program --- it was the editing commands that were the real doozies to program.

The Super Shape Draw disk also has programs to edit once a shape is done, continue later with an unfinished shape table, choose the shape table address or have it figured out for you, BLOAD the shape tables on the disk and view the shapes or BSAVE them onto a separate disk, etc.

Here are the vector codes:

	Vector	Code
Move Only	↑	ØØØ
	→	ØØ1 or Ø1
	↓	Ø1Ø or 1Ø
	←	Ø11 or 11
Plot and Move	↑	1ØØ
	→	1Ø1
	↓	11Ø
	←	111

8F. COMBINATION SHAPES (VECTOR, BLOCK, HPLOT)

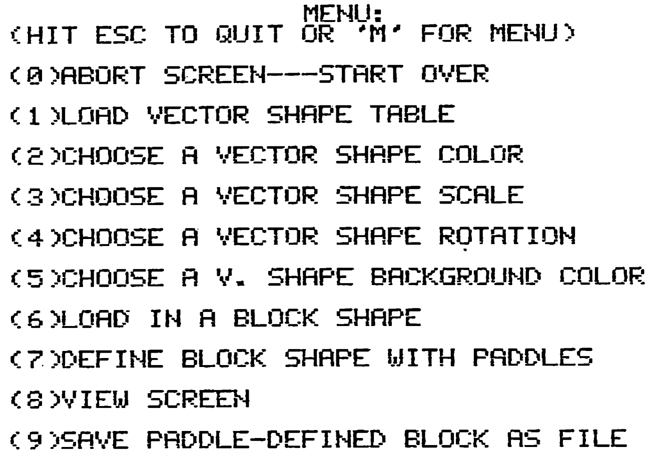
In the scanning program (4 of 28A) one may load in either vector shapes, block-shapes, or both, and then save any combination of shapes (created by loading them onto various sections of the screen) as 1 or more block-shapes.

You cannot put combinations of shapes into hplot-shapes, but with 5 of 28C you can put anything into vector shapes, and 4 of 28A will put anything into block-shapes. I could create programs to put other types of shapes into hplot-shapes easily enough, but for what purpose? The block-shape combinations are the most usable and convenient and vector shapes are super for rotating and scaling.

In order to get hplot-shapes to be combined with other types of shapes, and then turned into block-shapes, one must first use 7 of 28B to view an hplot-shape. Once you see the shape switch to 28A and go to menu and choose 4 on 28A, the scanner.

Here is the menu of the scanning program:

PRINTOUT #14



```
                   MENU:
<HIT ESC TO QUIT OR 'M' FOR MENU>
<0>ABORT SCREEN---START OVER
<1>LOAD VECTOR SHAPE TABLE
<2>CHOOSE A VECTOR SHAPE COLOR
<3>CHOOSE A VECTOR SHAPE SCALE
<4>CHOOSE A VECTOR SHAPE ROTATION
<5>CHOOSE A V. SHAPE BACKGROUND COLOR
<6>LOAD IN A BLOCK SHAPE
<7>DEFINE BLOCK SHAPE WITH PADDLES
<8>VIEW SCREEN
<9>SAVE PADDLE-DEFINED BLOCK AS FILE
```

Option 1 loads in vector shapes and option 6 loads in block-shapes. If you'd started with 7 of 28B and had an hplot-shape on the screen before entering 4 of 28A, then you'd hit SPACE BAR upon entry so the HGR command wouldn't end up re-initializing the hi-res screen, which would erase the hplot-shape.

Once into the program you continue filling the screen with various shapes until you've had enough. Then you go to subroutine #7 which is DEFINE BLOCK SHAPE WITH PADDLES. Use the paddles to move a dot-cursor to define the upper left-hand and lower right-hand corners of the rectangle that you want to define the perimeter of your block-shape. (If no dot is visible, turn the paddles counter-clockwise until it is.)

Once you've defined a shape go to option #9 to SAVE PADDLE-DEFINED BLOCK AS FILE (meaning a binary block-shape file with shapes whose #s are from 1 to 23, and with a length of 5888 or less bytes, starting at address $900 (2304 dec.).

Whenever you save a shape you'll be told beforehand to jot down the height and width that will be given on the screen (keep these #s with the shape # and shape table name).

After paddle-defining a block-shape you'll see the rectangle (with the HL rounded off downwards and the HR rounded off upwards so they're both at even bytes) you've defined. You'll be asked if it's okay. If you say it is you'll be asked if you wish to PDL-define others from the vector shape table you're on as entries for the block-shape table you're on. This allows creation of sequential block-shapes that are non-similar, like in a walking man sequence. If all the shapes in an animation sequence are the same but just moved over left or right incrementally, then use A of 28A or 1 of 28B rather than the scanner program.

Also, after scanning a shape you'll have the option to erase the screen before loading in the next shape, which is an option you'll need if you are saving non-similar shapes in a sequence. More on all this later.

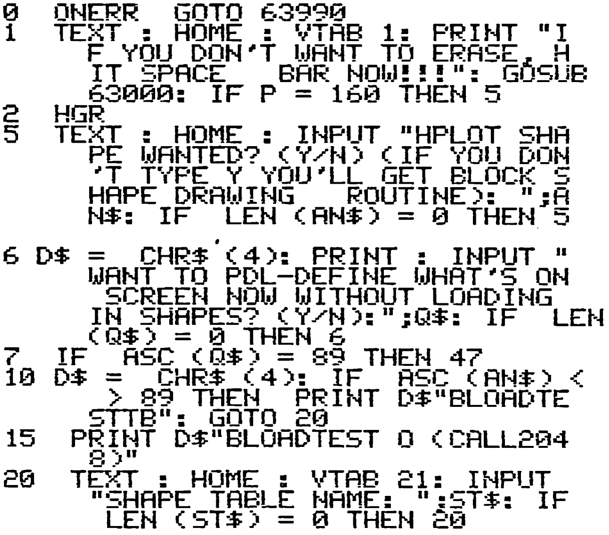
You'll also get asked to either give the # of shapes, and have the program use that (times 256) to determine the shape table length to BSAVE, or say that the last shape # given can be the last one in the table, in which case the last shape byte address dealt with minus the table address will be the length.

Shapes or scenes done in 2 of 28A may be combined with other shapes by going directly from 2 of 28A to 4 of 28A.

8G. CONVERSION TO VECTOR SHAPE

The following is a print-out of the program ZXZX, 38th on the catalog and directory of disk 28C. The menu calls the program CONVERT BLOCK-SHAPES TO VECTOR SHAPES, but if the truth be known, this program will convert anything (except your Granny's garter belts) into vector shapes:

PRINTOUT #999



```
0   ONERR  GOTO 63990
1   TEXT : HOME : VTAB 1: PRINT "I
       F YOU DON'T WANT TO ERASE, H
       IT SPACE   BAR NOW!!!": GOSUB
       63000: IF P = 160 THEN 5
2   HGR
5   TEXT : HOME : INPUT "HPLOT SHA
       PE WANTED? (Y/N) (IF YOU DON
       'T TYPE Y YOU'LL GET BLOCK S
       HAPE DRAWING   ROUTINE): ";A
       N$: IF  LEN (AN$) = 0 THEN 5

6 D$ =  CHR$ (4): PRINT : INPUT "
       WANT TO PDL-DEFINE WHAT'S ON
        SCREEN NOW WITHOUT LOADING
       IN SHAPES? (Y/N):";Q$: IF  LEN
       (Q$) = 0 THEN 6
7   IF  ASC (Q$) = 89 THEN 47
10 D$ =  CHR$ (4): IF  ASC (AN$) <
        > 89 THEN  PRINT D$"BLOADTE
       STTB": GOTO 20
15  PRINT D$"BLOADTEST O (CALL204
       8)"
20  TEXT : HOME : VTAB 21: INPUT
       "SHAPE TABLE NAME: ";ST$: IF
        LEN (ST$) = 0 THEN 20
```

```
30  PRINT : INPUT "SHAPE #: ";SN:
     IF SN > 23 OR SN < 1 THEN 3
    0
35  POKE 7,SN
40  IF  ASC (AN$) = 89 THEN 43
42  PRINT : INPUT "VT: ";VT: INPUT
    "VB: ";VB: INPUT "HR: ";HR: INPUT
    "HL: ";HL: POKE 252,VT: POKE
    253,VB: POKE 254,HR: POKE 25
    5,HL
43  PRINT : FLASH : PRINT "SWITCH
     TO SHAPE DISK!": NORMAL : GOSUB
    63000
45  PRINT D$"BLOAD";ST$:AD =  PEEK
    (43634) +  PEEK (43635) * 25
    6:L =  PEEK (43616) +  PEEK
    (43617) * 256: PRINT "ADDRES
    S: "AD: PRINT "LENGTH: "L
46  GOSUB 2000
47  POKE  - 16303,0: POKE  - 1629
    8,0: HOME : VTAB 1: PRINT "U
    SE THE PADDLES TO MOVE THE D
    OT TO THE  UPPER LEFT RECTAN
    GLE POINT. HIT PDL 0   BUTTO
    N. THEN MOVE THE DOT TO THE
    LOWER  RIGHT RECTANGLE POINT
    . HIT PDL 1 BUTTON.": GOSUB
    63000
48  POKE 232,250: POKE 233,0: SCALE=
    1: ROT= 64: POKE 250,1: POKE
    251,0: POKE 252,4: POKE 253,
    0: POKE 254,7: POKE 255,0
49  POKE  - 16304,0: POKE  - 1629
    7,0
50  HOME :P1 =  PDL (1): IF P1 >
    159 THEN 50
51  POKE 2300,1: POKE 2301,0: POKE
    2302,4: POKE 2303,0
55 P0 =  PDL (0): XDRAW 1 AT P0,P
    1:X% = P0:Y% = P1
60 P1 =  PDL (1): IF P1 > 159 THEN
    60
65  FOR QW = 1 TO 200: NEXT : HOME
    : VTAB 21: PRINT "X: "P0: PRINT
    "Y: "P1
70 P0 =  PDL (0): XDRAW 1 AT X%,Y
    %: XDRAW 1 AT P0,P1:X% = P0:
    Y% = P1
80 B0 =  PEEK ( - 16287): IF B0 >
    127 AND FL = 0 THEN FL = 1: GOTO
    100
85 B1 =  PEEK ( - 16286): IF B1 >
    127 AND SG = 0 THEN SG = 1: GOTO
    110
90  GOTO 60
100 VT = P1:HL = P0: PRINT  CHR$
    (7): IF SG = 1 THEN 120
105  GOTO 60
110 VB = P1:HR = P0: PRINT  CHR$
    (7): IF FL = 1 THEN 120
115  GOTO 60
```

```
120   XDRAW 1 AT P0,P1
155   HCOLOR= 3
160   HPLOT HR,VT TO HR,VB TO HL,V
      B TO HL,VT TO HR,VT
170   IF ZQ = 1 THEN  RETURN
175   VTAB 21
180   PRINT : INPUT "IS THE RECTAN
      GLE DONE O.K.? (Y/N):";A$: IF
       LEN (A$) = 0 THEN 180
185   IF  ASC (A$) = 78 THEN SG =
      0: HCOLOR= 0:FL = 0:ZQ = 1: GOSUB
      160:ZQ = 0: HCOLOR= 3: GOTO
      50
190 ZQ = 1: HCOLOR= 0: GOSUB 160:
     ZQ = 0
330 R = 1:FL = 0
340 Y = VT:X = HL - 1:LOC = 2304
350 X = X + 1: IF X > HR THEN B =
      2: GOSUB 500: GOTO 480
360   XDRAW 1 AT X,Y:B = 1: IF  PEEK
      (234) = 0 THEN B = 5
370 X = X + 1: IF X > HR THEN BB =
      16: GOSUB 500: GOTO 492
380   XDRAW 1 AT X,Y:BB = 8: IF  PEEK
      (234) = 0 THEN BB = 40
390 X = X + 1: IF X > HR THEN MB =
      128:FL = 1: GOSUB 900: GOTO
      460
400   XDRAW 1 AT X,Y:MB = 0: IF  PEEK
      (234) = 0 THEN  GOSUB 900: XDRAW
      1 AT X,Y: GOTO 360
410 MB = 64: GOSUB 900: GOTO 350
450 X = X - 1: IF X < HL THEN B =
      2: GOSUB 500: GOTO 380
460   XDRAW 1 AT X,Y:B = 3: IF  PEEK
      (234) = 0 THEN B = 7
470 X = X - 1: IF X < HL THEN BB =
      16: GOSUB 500: GOTO 400
480   XDRAW 1 AT X,Y:BB = 24: IF  PEEK
      (234) = 0 THEN BB = 56
490 X = X - 1: IF X < HL THEN MB =
      128:FL = 1: GOSUB 900: GOTO
      360
492   XDRAW 1 AT X,Y:MB = 0: IF  PEEK
      (234) = 0 THEN  GOSUB 900: XDRAW
      1 AT X,Y: GOTO 460
494 MB = 192: GOSUB 900: GOTO 450

500 Y = Y + 1: IF Y > VB THEN 100
      0
505 R =  NOT R
510   RETURN
900 BY = B + BB + MB: POKE LOC,BY
      :LOC = LOC + 1:B = 0:BB = 0:
      MB = 0
905   IF FL = 1 THEN FL = 0: GOSUB
      500
930   RETURN
1000   POKE 232,252: POKE 233,8: ROT=
      0: SCALE= 1
1005   POKE LOC,0: POKE LOC + 1,0
```

```
1010  XDRAW 1 AT 140,79
1015  HOME :SG = 0:FL = 0
1020  VTAB 21: GOSUB 63000
1025  TEXT
1030  PRINT "ADDRESS OF 1-SHAPE I
     NDEX: 2300 (DEC.)   ADDRESS
     OF START OF SHAPE 2304":LN =
     LOC - 2304:LT = LOC - 2300: PRINT
     "LENGTH OF TABLE: "LT: PRINT
     "LENGTH OF SHAPE: "LN
1040  GOSUB 63000: HOME : VTAB 1:
      INPUT "DO YOU WANT ANOTHER
     CONVERSION? (Y/N):";AS$: IF
      LEN (AS$) = 0 THEN 1040
1050  IF  ASC (AS$) = 89 THEN 20
1055  PRINT : PRINT "HIT RESET AN
     D THEN BSAVE A VECTOR SHAPE
      IF DESIRED.": GOSUB 63000
1060  PRINT D$"RUNMENU"
2000  IF  ASC (AN$) <  > 89 THEN
      CALL 2116: RETURN
2005  HCOLOR= 3
2010  CALL 2048: RETURN
63000  PRINT : PRINT "(HIT ANY KE
     Y TO CONTINUE):": PRINT
63010 P =  PEEK ( - 16384): IF P >
     127 THEN  POKE  - 16368,0: RETURN

63020  GOTO 63010
63990  POKE 216,0
63991  ONERR  GOTO 63990
63992  IF  PEEK (222) = 254 THEN
      RESUME
63995  GOTO 0
```

Line 47-190 paddle-defines a block of memory as the shape you want to save as a vector shape. It does this in one of 3 ways:

1) If you have pictures or shapes already on the screen on entering the program, you may use part of the screen as your vector shape definition --- any rectangular area.

2) You may load an hplot-shape onto the screen.

3) You may load a block-shape onto the screen.

Line 330-930 algorithm for changing raw screen bytes into vector shape bytes. Here's how it works (it's in Applesoft and takes around 10 seconds per square inch of area --- I was too lazy to figure it out in assembly, mostly because I've never really had any **need** to turn any type of shape into a vector shape):

The idea is to use the collison counter! I've never seen Paddle Graphics work, but since it creates rotatable shapes, I'd wager that this is the type of routine used in that package. It's about the easiest method I know of.

So what do I **do** with the collision counter? Simple. I first draw a one-dot shape (line 48 POKES in the shape table --- the 7 in dec. 254 is the dot). I notice that this puts the collision counter at 1. Then I stick the paddles "into the game" --- and I learn that if I control the whereabouts of the dot-shape with the paddles and ram the dot into a shape, it changes from 1 to 0. Then I figure out that this is because I'm EORing with the shape and 2 ones make a 0 in EOR, and a dot-shape hitting other shapes is combining 2 ones.

So I say to myself, okay, if I get such predictable results when I walk upon other shapes with my dot-shape, why not use it as a dot by dot indicator in examining a block of memory? Why not have the collision counter (peek(234)) be my "I found an ON bit" flag? By knowing which dot is at what X,Y coordinates (279 by 191) I could easily translate this very specific information into vector shape information. All one needs to know is when to move and when to plot. And when a bit (corresponding to X,Y coordinates) is off, I move, and when it's on I plot. The only vector bytes I needed were ones made up of combinations of the following binary groups, each group having a specific decimal value as shown, and each group corresponding to either column A or B or C in the shape table chapter in the Applesoft manual (pgs. 92-94):

Section	Binary Configuration	Meaning	Decimal Value
1) C	00	Ignore	0
2) A	001	→	1
3) A	010	↓	2
4) A	011	←	3
5) A	101	•→	5
6) A	111	←•	7
7) B	001	→	8
8) B	010	↓	16
9) B	011	←	24
10) B	101	•→	40
11) B	111	←•	56
12) C	01	→	64
13) C	10	↓	128
14) C	11	←	192

The 14 combinations are all it took; there was no reason for upmoves or vertical plots. The program scans through the defined width of the "vector-block" shape (HL to HR) at VT and reverses direction and moves down a line and does it again, and reverses again at the end of that line, etc., until finally the VB line is done.

Every time the collision counter is 0 (which means my dot-ONE met a shape ONE), I use the decimal #s that signify PLOT --- otherwise I use the MOVE ONLY #s. Once I've got 3 columns worth (sometimes column C is a Ø because it's found that there are 3 or more ON bits in a row, and only 2 plotting vectors per byte are permitted, since plots take 3 bits) I add up the #s (see line 9ØØ) and POKE them into a shape table I build whose index address is 23ØØ and whose shape address is 23Ø4 (dec.).

You get to watch the dot in action; it inverses the entire shape. Once it's through you'll see address and length (jot them down!) and the new vector.shape will be drawn at X,Y coords. of 14Ø, 79. If you wish to save the shape, hit RESET when told to and BSAVE (Shape), A23ØØ, L (length). To use the shape, with whatever rotation you desire and whatever scale, BLOAD it and do HGR and POKE 232, 252 and POKE 233, 8 and do ROT and SCALE and HCOLOR and the DRAW or XDRAW.

R=NOT R is merely a direction-changing flag, that toggles to the opposite setting from where it's currently at every time R=NOT R is executed.

The uses for this program are many:

1) **Change the rotation or scale of a block-shape** by saving the original block-shape as a vector shape, and then going to 6 of 28D to scan and save the block-shape in this new scale or rotation. You needn't even bother to load the shape in 6 of 28D or save it after resetting in 5 of 28C. All you need to do after resetting is ROT=16:SCALE=2. (or whatever):XDRAW 1 AT 3Ø, 9Ø (or whatever). Then POKE-163Ø4, Ø and POKE-16297, Ø to see screen --- if okay then insert 28D and RUN MENU (Return). Go to 6 of 28D and hit Space Bar and paddle-define (option #7) the new shape rotation and save it. Neat, huh? This little utility was a full afternoon's work. The slightly tricky lines were lines 4ØØ and 492. I won't say why --- there's a cute project for you.

2) 5 of 28C creates vector shapes that are always referenced from their reference point of **the upper left corner.** Rotations are around this point. This may come in handy.

3) You may need either hplot-shapes or block-shapes to be larger. The way to do this so it looks good is to DRAW (not XDRAW) **the square of the scale #** of shapes. If scale is 2, DRAW at 98, 98 and 98, 99 and 99, 98 and 99, 99. This will fill in the scaling-caused gaps.

4) To get a rotated hplot shape, save it as a vector shape, rotate it and save it, and have it on-screen or hard-copy print-out when you redo the rotated version in 8 of 28B. (Actually, it's extremely fast and easy to draw an hplot-shape --- you needn't bother to do all this using of 5 of 28C, etc. Merely calculate or estimate rotated coordinates via 6 of 28B and draw in 8 of 28B.)

5) To put a shape saved in 5 of 28C into an existing vector table, BLOAD it at previous shape's last location (a Ø) + 1, after saving the 23Ø4 (not the 23ØØ) version in 5 of 28C.

SHAPE SEQUENCE CREATION 9

9A. VECTOR SHAPE SEQUENCE

In a program in the Super Shape Draw and Animate Package, you get to start with one vector shape and convert it into a whole sequence of different shapes, each one different from the next in just the ways necessary for good animation.

These shapes can be used in vector shape animation routines, such as the Animation Demo on the Super Shape Animate disk, or they can be converted into block-shape sequences via 4 of 28A. More on block-shape sequences later.

Program 3 on 28A will allow you to view the vector shapes you're going to be putting into animation sequences. Program I on 28A will allow a detailed examination of any vector shape. Programs G and H (on 28A) will demonstrate 1 page and 2 page vector shape animation.

This graphics system will let you create and use animation sequences of both block-shapes and hplot-shapes. With vector shapes you need to create and use the sequences via our Super Shape Draw and Animate Package or some other animation package, but you may also use our scanner program (4 of 28A) to convert these vector shapes into block-shapes, which animate faster than vector shapes, when good machine language routines are used; and even faster yet with the use of YTABLE. More on that later.

In some animation, only one shape is used --- it's drawn and erased over and over, with or without hi-res graphics page flipping. In vector shape animation of this kind one need only change a coordinate slightly between erase and draw routines. In block-shape animation of this kind a shape may not move less than 1 byte at a time unless the shape uses logical shift animation, which you'll find on 28B.

If you make a 7-shape sequence out of the starting shape and each one is saved only a bit or 2 away from the last, then you can turn a block-shape into a sequence that needs no logical shift animation in order to animate it. All that's needed are regular block-shape drawing routines with properly done sequential drawing and erasing routines. Even though a block-shape can't move less than one byte sideways without logical shift; a sequence of 7 block-shapes does 1-2 bit moving admirably. There are programs to create such sequences on both 28A (program A) and 28B (program 1).

If you've never done any animation I'd recommend using simple vector shape movements, using XDRAW for both draw and erase, until you get the hang of it. Then go on to block-shapes and hplot-shapes.

9B. BLOCK-SHAPE SEQUENCES

There are a lot of programs in this graphics system that are based upon good block-shape sequences. The 28A programs are 6, B, C, D. There are only hplot or logical shift programs on 28B in the animation area. None of these require block-shape sequences. (Don't forget, though, that 1 of 28B is an automatic shape sequence creator that starts with block-shapes and ends with block-shapes; while A of 28A is the same only it starts with vector shapes and ends with block-shapes.)

On 28C there is a block-shape sequence-animation routine using YTABLE. It's #4 on the menu. It's hard to believe that the shape is moving only 1 bit sideways per move. A shape seems to make it across the screen in about 2.8 seconds as opposed to 3.5 second without the use of YTABLE. (YTABLE's weakness is the amount of room it takes up --- it requires either two or three ¾-page tables; which can take up up to nearly 2½ pages.) This is the fastest **1-bit animation** in our system, and F on 28A is our fastest **1-byte animation**, which is something program 3 of 28A allows you to try out with inputs. 28C is an unprotected disk (it also has all the source-code files).

Now, here are the block-shape sequence tables on these disks and what programs to try them in:

addr:,length:	28A:	use w/programs:	# shapes	heightxwidth (#5 is 15x3)		steps	sequence
A2304,L2370	MANA	3,6,D,F of 28A	10	21	2	1	1,6,2, 7,3,8, 4,9,10
A2304,L1624	MANB	D of 28A	7	21	3	1	1-7
A2304,L1646	MANC	D of 28A	7	21	4	2	1-7
A2304,L1792	Q3	B,C,D of 28A	7	18	5	1	1-7
A2304,L1792	Q4	D of 28A	7	21	8	1	1-7
A2304,L1792	Q5	D of 28A	7	41, but say 21 21	8	2	1-7
	28B:						
A2304,L2370	MANA	3,6,D,F, of 28A	10	21	2	1	1,6,2, 7,3,8, 4,9,10
A2304,L4096	SEX	D of 28A	16	#1 & #2 are 27X4 27	5	3-9, 10-16	
A2304,L1792	AA	D of 28A	7	16	5	1	1-7
A2304,L1792	SQSEQC	D of 28A	7	32	6	2	1-7
	28C:						
A2304,L1792	Q3	4 of 28C	7	18	5	1	1-7

Here are the menu names of all **block-sequence** related programs in this system, and the names of the unprotected source-code files (on 28C only) and machine language files they relate to (the mach. lang. files are driven by these Applesoft programs):

Block-shape Table (seq.):	28A:
(any)	A) Automatic Block-Shape Sequence Creator **Source:** TESTTB! **Mach. Lang.:** TESTTB, A2Ø48, L14Ø
Q3	B) 2-Page Flipping Block-Shape Demo In Assembly **Source:** TEST E **Mach. Lang:** TEST E (CALL36934), A36864, L288
Q3	C) 1-Page Block-Shape Demo In Assembly **Source:** TEST D **Mach. Lang.:** TEST D (CALL2186), A2Ø48, L21Ø
MANA MANB MANC Q3 Q4 Q5 SEX AA SQSEQC	D) 2-Page Flipping Block-Shape Demo In Assembly; With Inputs **Source:** TEST F, TEST G **Mach. Lang.:** TEST F (CALL36934),A36864,L324 TEST G (CALL36934),A36864,L342
	28B:
(any)	1) Automatic Block-Shape Sequence Creator --- Starting With Block-Shapes **Source:** TEST H **Mach. Lang.:** TEST H (CALL2186),A2Ø48,L224
	28C: (unprotected) (has all "TEST" programs, both source and mach. lang.)
Q3	4) YTABLE Test (Vert. Addr. Locator) **Source:** TEST E **Mach. Lang.:** TEST E (CALL36936),A36864,L288 YTABLE,A$1DØØ,L$28Ø

You may try other block-shape tables in some of the programs or make tables of your own and try them in various programs.

Chapter 8B has already supplied you with a source-code list-out for TEST TB! All TEST TB! is is a short scanner program, for the creation of a block-shape from a previously paddle-defined area of the hi-res screen, and a short drawing program, that retrieves block-shape data from BLOADed binary file data arrays and puts it into a block composed of some of the hi-res screen bytes ($2000-$3FFF or $4000-$5FFF). The programs start at the lower right corner of a block shape and insert or retrieve data bytes, one at a time, moving left until the HL byte is done. Then the program moves up a line and gets the next line's data. This continues until the last line (VT) is done.

Make sure you're familiar with how this scanning and drawing happens before going on.

Program A in 28A uses vector shapes to create block-shape sequences of 7 shapes. If you have no vector shape sequences, you may create block-shape sequences with 1 of 28B by using other block-shapes.

The object of a block-shape sequence is to move block-shapes less than one byte per move, since 7-dot moves (1 byte) are too large to look smooth, although they are quite fast.

The normal step-size for block-shape sequences is either 1 bit (or "dot") or 2 bits.

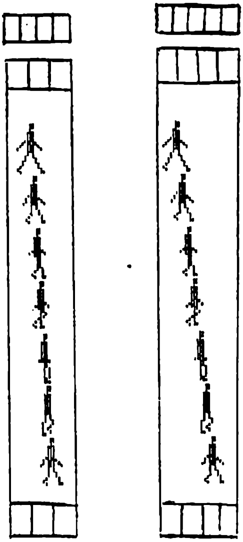
The advantage of 1-bit moving is that it's as smooth as you can get, especially if you use page-flipping. The disadvantage is that colors will change like crazy as your shape traverses the screen, unless you're in black and white only.

The advantage of 2-bit moving is that it's still quite smooth but colors will stay the same in your shapes---there will be no color problem. A colored shape shifted one dot sideways will no longer be the same color. This works to your advantage for creating certain types of effects, but is usually a "pain in the assembler". Some colors work at even horizontal screen coordinates and others work at odd ones. It's an Apple fact of life. That's why resolution in color graphics on an Apple is limited to 140x192. But this isn't a complaint---I love the Apple and accept its limitations without any tears or harsh words.

In using A of 28A, ask for steps of 2 bits per move, if you're using color. You might use this for black and white also and simply forget 1-bit moving --- it's more convenient than using both methods.

Let's look at a 1-bit moving sequence and a 2-bit moving sequence:

PRINTOUT #15



The first sequence is the contents of the block-shape table named MANB and the second sequence is the contents of the block-shape table named MANC.

Notice that MANB shapes move 1 bit from 1 shape # to the next, while MANC shapes move 2 bits. If you try out MANB in D of 28A you'll see that 1-bit movement is sometimes inferior to 2-bit movement in another way besides color irregularity: some sequences of non-similar shapes that represent walking or running or whatever need steps larger than 1 bit. MANC looks much better than MANB when run as a 1-7 sequence in D of 28A with height of 21 and width of 4 (the MANC width is 4, while the MANB width is 3)(Use hi. delay of 7Ø and lo. delay of 255).

Let's look more closely at the above graphics print-out. The **width you give** with MANB is 3 and the width you give for MANC is 4, which means the **actual** width of each of these is 1 more; i.e.: MANB is actually 4 wide and MANC is actually 5 bytes wide. Remember, if you input that you want MANC shapes to go from 25 to 29 (horizontal bytes) then 25,26,27,28,29 is obviously 5 bytes wide, however, 29 minus 25 is 4. The discrepancy has to do with the fact that from 25-29 is in the **inclusive** sense as far as the actual byte width, whereas in the **width to input** area 25 to 29 is referring to **distance between** these 2 bytes --- 4. If this is confusing, reread the previous chapters.

In the above print-outs there are 2 sets of grids: immediately above and below the shapes are boxes each of which represent one byte wide. They depict the **width to give** for each sequence. These grids do not represent a true picture at all---the block-shapes are all 1 wider than these grids depict. They are the grids you would imagine if you were still confused by actual width vs **width to give.** However, above the top grids are other grids which **are** correct. These sequences **are** 4 and 5 bytes wide, respectively.

You may wonder why a shape that's about 2½-3 bytes wide is being given 4 or 5 actual bytes---isn't that a time-waster? Well, a sequence needs to hold all 7 shapes, and all 7 shapes must fit within the given block of bytes. You see, when a "sequence" is actually used in an animation routine using vector shapes that are all the same, then you merely draw and erase the shape, moving it over each time an erasure is completed. But when a vector shape sequence is composed of dissimilar shapes, then not only do you draw and erase and move over, but you also must change **which shape #** is being drawn/erased.

Block-shapes sequences are similar to this last situation except for the moving-over aspect. A block-shape sequence increments (or decrements, if it's moving leftwards) its horizontal byte coordinate only once per sequence, not once per shape. In other words, within the block there must be room for 7 shapes that move over 1 or 2 bits per step. This means that if a shape was 17 bits wide and took up 3 bytes of width (3x7) then with 1-bit steps it would be 23 bits wide and take up 4 bytes. If it used 2-bit steps it would be 29 bits wide and take up 5 bytes. These figures all represent approximately what's true for MANB and MANC.

We round off upwards to the nearest multiple of 7, so 29 takes up 5 bytes because 4x7=28 in which case we might lose the edge of the shape, and 5x7=35, which is the nearest higher multiple of 7.

Now notice how either of the above sequences would appear to move if the shapes were to be sequentially drawn and erased. The man in MANC would appear to move with bigger steps but at the same duration per step. (In actual fact MANB appears to be sliding on ice or walking on a treadmill because the step-sizes are considerably too small. MANC with 2-bit steps is fairly realistic---perhaps a 2½-bit (impossible without extremely sophisticated use of 56Øx192 resolution graphics) step size would be ideal.)

When trying out MANB and MANC in D of 28A, I would recommend that when you're asked for hi byte of delay loop you choose 7Ø, and when asked for lo-byte of delay loop you choose 255. This will give the most realistic performance. A delay loop such as that given means that 17,85Ø loopings are performed in between drawing and erasing each shape!

I'm aware that these 2 delay-loop inputs are not regular hi-byte and lo-byte inputs at all. In fact, even if you choose 255 for the lo-byte, you'll still not have a lo/high byte ratio here. A standard lo-byte is 1/256 of what the hi-byte bytes stand for. In choosing 255 for the so-called lo-byte we get very close to a correct lo/high ratio, but no cigar.

The truth is, in our delay loops the total looping equals the "hi" delay loop byte times the "lo" delay loop byte. Each could be ten, or 99, or 1, or 255, or they could be dissimilar. (By the way, Ø would have the same effect as 256 (illegal) in these delay loops since these numbers get decremented BEFORE the BNE opcode checks to see if it's zero yet. And a decremented Ø will give you a 255 every time. In actuality a decremented Ø gives you a binary number of 11111111, which in two's complement representation is -1, which is logically what it should be since Ø minus 1 equals -1. However, for our purposes let's look at the straight binary number 11111111 as 255. The largest delay loop available, then, in our routines, is 256X256=65536, which you'd get, in effect, by inputting Ø's into each delay loop byte. The shortest delay, resulting in 1 run-through and no loop-backs, in the delay loop, would be accomplished by inputting 1's in each delay byte. This would not be true of **Applesoft** delay loops such as might be inputted in the animation routine in 3 of 28A. We're considering such programs as D of 28A at this time.)

So the 1st delay-loop byte is "high" only because it specifies how many times to loop through the "low"-byte loop, a loop which may be run through, until decrementing results in zero, anywhere from 1 to 256 times. Think of H times L, in this usage of high and low, rather than (H times 256) plus (L times 1). Sorry I took such devastating liberties with convention, semantics, and the lady down the stre----ah, but that's another story.

If you've gotten this idea of block-shape sequences changing horizontal coordinates only once per sequence (at the end), then you'll be imagining shape #1 to #7 drawing and erasing sequentially while the block-shape is at coordinates (example) HL=7, HR=1Ø, VT=7Ø, VB=91, and then the horizontal coordinate shifts by 1 and the sequence is again run through at HL=8, HR=11, VT=7Ø, VB=91. The above would be for MANB. For MANC the same exact thing would happen except that coordinates such as HL=7, HR=II, VT=7Ø, VB=91 would change by HL and HR being incremented by 2 so that (for the next sequence) you ended up with HL=9, HR=13, VT=7Ø, VB=91. The reason the horizontal coordinates go up (or down if you're moving leftwards) by 2 is that this sequence was built with a 2-bit movement between shapes, which means that a 2X7=14-bit actual displacement will have taken place in the course of one sequence run-through. So if you moved the horizontal coordinate only 1, you'd end up with a walking mode of "2 steps forward, 1 step back", etc., in effect. Again consult the shape sequences illustrated to confirm this. Or try to input a "step" of 1 for MANC. You can also see by all this the value of clearly recording securely all such data as shape table name, # of shapes, height, width, and step-size!

If you're wondering why it is that an Applesoft FOR-NEXT loop enclosing no commands would be about 1 second long for every 800 loops while animation at an approximate rate of 11 frames per second would have 17,850 loops (for delay) per shape and 196,350 per second and 124,950 per 7-shape sequence, the answer is that machine language is the computer's language, whereas Applesoft must undergo the time-consuming process of getting interpreted as computer-understandable commands before it makes things really happen.

In practice, a program would be more likely to have IF-THENS or ON-GOTO's or ON-GOSUB's, in Applesoft, or BNE, BMI, BPL, BEQ, BVC, BVS, BCC, BSC in machine language than simple delay loops. A good hi-res game is going to spend a lot more time wondering if the score has reached so and so, or if there's been a collision counter change, or if hi-res coordinates of certain shapes have reached so and so, than it will spend in do-nothing delay loops.

You see, when a delay is needed for animation or game speed/control, a programmer sees this most often as the fortunate creation of space and time for dynamic programming, rather than a time for the computer to kill time. A generality that would apply in most cases is that whenever there's more time than is needed for what's happening, or whenever speeds of any aspect of a game will be excessive unless slowed down, a good programmer thinks of how many dynamic/intriguing qualities can be added to his/her game with the extra time, while a bad programmer would throw in the delay loop with no thought for using the time to greater advantage. This doesn't mean there's something wrong with my programming since I've incorporated straight delay loops into most of the animation routines. In routines such as are included in this graphics system, one needs straightforward delay-loop options. Later when actual game-creation begins, it will be the sole responsibility of the programmer to replace (partially or wholly) the delay loops with more dynamic programming.

Does a 2-bit shape movement always mean a 2-byte (hor.) step-size? Yes, as long as there are to be 7 shapes per sequence. There's not usually reason for longer sequences but our sequence-creators let you choose sequence length. If for some reason you wanted to make a modification to sequence-creation programs in this system you'd have to do it the "hard way". (You may list programs on 28A or 28B by use of E or F on 28B, but you may not make program changes without retyping the programs (this refers to programs you go to from the MENU---they're in Applesoft). You may change the source-codes for any of the assembly routines by EXECing test-file-type LISA source-code files and then changing, assembling, and re-writing them. You may change the object code (the binary code output by an assembler often stored in binary files that get BLOADėd) by monitor commands (see reference manuals for the Apple) and BSAVING afterwards if you wish to make it permanent, or by BSAVING the object codes assembled in LISA right after the reassembly is complete, when you've changed a source-code file.)

A 1-bit shape movement with a 7-shape sequence would require stepping 1 byte after each sequence.

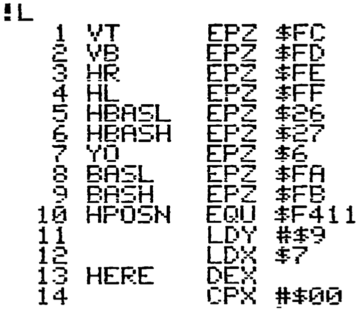
Again look at the 2 sequences in the illustration. It should now be clear that for the "drawing" of the seven shapes in the sequence the exact same block of bytes on the hi-res screen is dealt with. Think of it like this:

					Block drawn Into:
Shape #	**draw**	**data Is from**	**erase**	**data Is from**	**HL,HR,VT,VR**
1	data array 4 by 22	$900-$957	data array 4 x 22	$900-$957	7, 10, 70,91
2	" "	$A00-$A57	" "	$A00-$A57	" "
3	" "	$B00-$B57	" "	$B00-$B57	" "
4	" "	$C00-$C57	" "	$C00-$C57	" "
5	" "	$D00-$D57	" "	$D00-$D57	" "
6	" "	$E00-$E57	" "	$E00-$E57	" "
7	" "	$F00-$F57	" "	$F00-$F57	" "

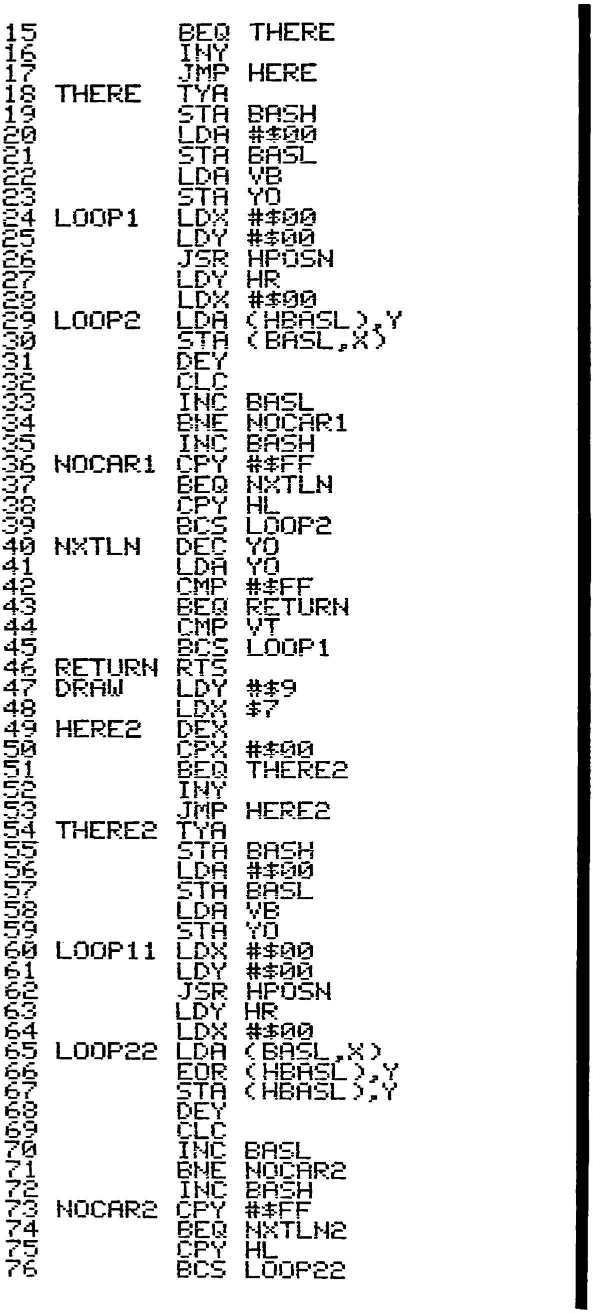
It is only **after** all these shapes are drawn and erased that the horizontal coordinate is altered. Exactly how are the drawing/ erasing operations performed? Simple---the miracle of XDRAWING, which is a logical operation called EOR (exclusive OR) in assembly, is that if you draw a thing once, there it is, and if you repeat the procedure and draw it again in the same place it disappears completely. Two white dots equal a black dot, so when you draw white on a black screen, you get a white shape (this applies to vector shapes and block-shapes but not hplot shapes), but when you draw white again (if you're EORing) on that white shape, it simply erases the shape and the screen is once again empty. So in the sequence above a shape is drawn, then redrawn (erased), and then the next shape is drawn and redrawn (erased), and so on. After #7 is drawn and erased the horizontal coordinates are moved over.

It should be noted that when hi-res page flipping is involved (using both pages) the complexity involved in what gets drawn or erased increases considerably. But for now let's concentrate on 1-page (HGR) animation with rightwards movement. Here is the assembly routine (called TEST D) for C of 28A, which is the One-Page Block-Shape Demo In Assembly.

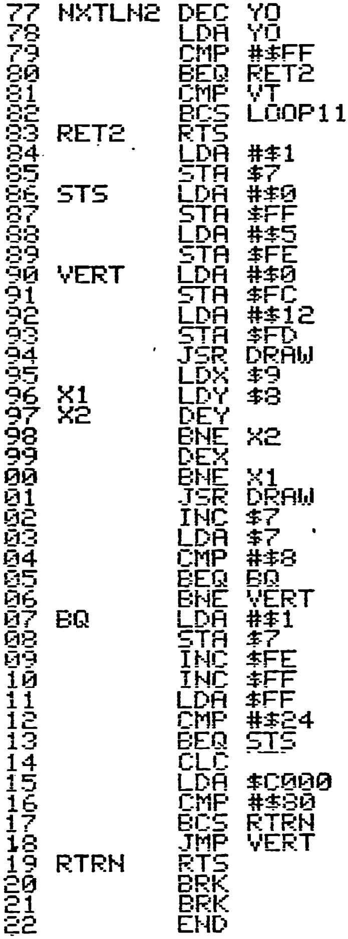
PRINTOUT #16



```
!L
  1 VT      EPZ $FC
  2 VB      EPZ $FD
  3 HR      EPZ $FE
  4 HL      EPZ $FF
  5 HBASL   EPZ $26
  6 HBASH   EPZ $27
  7 YO      EPZ $6
  8 BASL    EPZ $FA
  9 BASH    EPZ $FB
 10 HPOSN   EQU $F411
 11         LDY #$9
 12         LDX $7
 13 HERE    DEX
 14         CPX #$00
```



```
15          BEQ THERE
16          INY
17          JMP HERE
18 THERE    TYA
19          STA BASH
20          LDA #$00
21          STA BASL
22          LDA VB
23          STA YO
24 LOOP1    LDX #$00
25          LDY #$00
26          JSR HPOSN
27          LDY HR
28          LDX #$00
29 LOOP2    LDA (HBASL),Y
30          STA (BASL,X)
31          DEY
32          CLC
33          INC BASL
34          BNE NOCAR1
35          INC BASH
36 NOCAR1   CPY #$FF
37          BEQ NXTLN
38          CPY HL
39          BCS LOOP2
40 NXTLN    DEC YO
41          LDA YO
42          CMP #$FF
43          BEQ RETURN
44          CMP VT
45          BCS LOOP1
46 RETURN   RTS
47 DRAW     LDY #$9
48          LDX $7
49 HERE2    DEX
50          CPX #$00
51          BEQ THERE2
52          INY
53          JMP HERE2
54 THERE2   TYA
55          STA BASH
56          LDA #$00
57          STA BASL
58          LDA VB
59          STA YO
60 LOOP11   LDX #$00
61          LDY #$00
62          JSR HPOSN
63          LDY HR
64          LDX #$00
65 LOOP22   LDA (BASL,X)
66          EOR (HBASL),Y
67          STA (HBASL),Y
68          DEY
69          CLC
70          INC BASL
71          BNE NOCAR2
72          INC BASH
73 NOCAR2   CPY #$FF
74          BEQ NXTLN2
75          CPY HL
76          BCS LOOP22
```



```
77  NXTLN2  DEC YO
78          LDA YO
79          CMP #$FF
8Ø          BEQ RET2
81          CMP VT
82          BCS LOOP11
83  RET2    RTS
84          LDA #$1
85          STA $7
86  STS     LDA #$Ø
87          STA $FF
88          LDA #$5
89          STA $FE
9Ø  VERT    LDA #$Ø
91          STA $FC
92          LDA #$12
93          STA $FD
94          JSR DRAW
95          LDX $9
96  X1      LDY $8
97  X2      DEY
98          BNE X2
99          DEX
1ØØ         BNE X1
1Ø1         JSR DRAW
1Ø2         INC $7
1Ø3         LDA $7
1Ø4         CMP #$8
1Ø5         BEQ BQ
1Ø6         BNE VERT
1Ø7 BQ      LDA #$1
1Ø8         STA $7
1Ø9         INC $FE
11Ø         INC $FF
111         LDA $FF
112         CMP #$24
113         BEQ STS
114         CLC
115         LDA $CØØØ
116         CMP #$8Ø
117         BCS RTRN
118         JMP VERT
119 RTRN    RTS
12Ø         BRK
121         BRK
122         END
```

Notice that the first 83 lines are merely TEST TB!, which is the routine for scanning followed by the routine for XDRAWING. After that comes the animation routine. Let's go through it together: (Consult an assembly language book where needed.)

Line 84 puts a 1 in the accumulator and line 85 transfers it from the accumulator to the zero page address $7. We'll almost always be using $7 to store the "current shape # being dealt with". This means that you'll need to POKE 7, shape # in your own programs, or in my programs I'll have to make sure $7 always has the correct shape #.

Line 86-89 puts Ø into $FF, which we'll usually use to store HL, and 5 into $FE, which will be HR. Lines 90-93 put Ø into $FC (for VT) and 18 ($12) into $FD (for VB). Whenever HL moves X amount, HR must also move X amount, and the same goes for VB and VT, since the size of the block must stay the same, regardless of where the screen location coordinates are.

Line 94 is like a BASIC GOSUB, in that it jumps you to line 47 and takes you through line 83 where the RTS is like a BASIC RETURN command, so you return to the line after 94; i.e. 95.

Lines 95-100 is the delay loop. It works like this: $9 holds the "high" byte and $8 holds the "low" byte of the delay loop. The number in $9 gets put into the X register in 95. Then the # in $8 gets put into the Y register in line 96. In 97 the Y register is decremented by 1 and in 98 the computer checks to see if the results of the previous line resulted in a zero. If it didn't then the routine branches back to line 97 and decrements Y again. This continues until Y is 0 and line 98 finds that 0 and therefore does NOT branch, but continues to line 99, which decrements X. In line 100 if X is now zero the delay loop is finished and the routine goes on to line 101 where erasing happens (by redrawing via EOR). If X > 0 then the routine goes all the way back to line 96 and Y is again loaded with $8 and decremented all the way down to 0. Then X is again decremented and checked to see if it's 0. Eventually X will be 0 and the delay loop is done for now. It's easy to see that the "high" # in X controls how many times the Y register has to go through its entire loop (from # in $8 to 0).

So after the delay (that keeps the shape from looking flickery by keeping it on the screen for a bit) the erasing happens at 101. Then in 102-103 we put the next higher shape # in $7, and we see if it's an 8 in 104. If it is then 105 tells us to go to 107, skipping 106, which takes us back to drawing and erasing again. In 107, since we've found that we've incremented beyond the # of shapes in our sequences (7), we reload $7 with a 1. We then check to see if we're getting too close to the edge of the screen in 111 and 112, after moving over the horizontal coordinates in 109 and 110. In 113 if our HR has reached 36 ($24) then we start everything all over again at line 86. If HR < 36 then we continue the drawing and erasing processes that begin at 90.

Lines 109-110 are the way the horizontal coordinates get increased, since HL is in $FF and HR is in $FE (see lines 1-10, which equate certain labels with addresses).

How does 103-106 work? Well, 103 puts the current shape #, learned from $7, into the accumulator. Line 104 compares the accumulator with the number 8. Line 105 says that if an 8 was found in $7 then skip 106 and go to the line labelled BQ, which is 107. In truth, what a CMP opcode does is subtract the data (8) from the # in the accumulator. The result isn't stored but status flags get set. One of these flags is Z. The Z flag, if set, means the result was 0. BEQ is an instruction that looks at the Z flag and branches to the given line **If** the Z flag is set --- set means equal to 1. (One is often used to mean yes, set, on, or true while 0 often means no, not set, off, or false, in computer logic.)

In an actual binary file, there are neither line numbers nor line labels, so the way branching is accomplished is via displacement #s which tell the routine that if certain conditions are met, such as Z=1, then it's time to branch forward or backward a certain # of bytes. Displacments may be from +129 to -126.

The BNE in 1Ø6 means that branching happens only if the last result of an operation was NOT zero. Whoever invented these computer instruction languages was surely a real sweetheart --- these neat little 3-letter commands are so irresistibly **cute** that I can hardly stand it. So if they seem formidable or nebulous at first, don't sweat it: soon you and opcodes will be the best of friends.

Line 118 has an unconditional **jump** back to line 9Ø. Line 12Ø and 121 will stick double zeroes in the last 2 locations of a BLOADed binary file, to make the end of the routine easier to spot. Line 122 is simply a LISA command saying "that's all, folks"---if you don't use it you get an error in assembling.

So why have I ignored lines 114-117? Because these commands are not related to animation, they are simply a convenient way for you to get out of this machine language program and get back into the Applesoft program that called it. In line 114 you clear the carry bit (set it to Ø) with a CLC. The C (carry) bit is much like the Z bit. Both are flags found in the status register that act as signals and are flip-flopped either **on** (1) or **off** (Ø). But the carry bit signifies that a result wasn't contained within the 8 bits in the accumulator so it needed a 9th bit in that it holds the 1 that's "carried" in an addition, or if you're subtracting, it must be **set** with SEC, and it will indicate a **no-borrow** condition until it gets set to Ø (cleared) by being used in a borrow.

In line 115 you load the byte found in $CØØØ. This is the Keyboard Data Input address. You're loading this byte into the accumulator so you may test it. The way you test it is to see if the 8th bit is set or not. If it is, then the number represented by the $CØØØ byte will be 128 or greater. $8Ø is 128 in dec., which is why line 116 compares the byte in $CØØØ (which is now also in the accumulator) with #$8O. The # in front of $8Ø means you're talking about the immediate data $8Ø or 128, rather than the byte found at address $8Ø. It discriminates between "contents of address" and "data immediately used."

The highest bit (#7) of the $CØØØ byte is a kind of flag that's used to indicate whether or not the keyboard's been hit. The flag is a 1 if keyboard hitting has occurred and a Ø if it hasn't. So if you find out by CMP #$8Ø that the keyboard's been hit, then you get sent to line 119 where you come back from the Applesoft CALL that got you into the machine language routine in the first place. The C of 28A program will then allow you to try other delay loops or return to the MENU.

The 1st 7 bits in $CØØØ represent the ASCII code of the key which was most recently pressed. A dec. representation of that code will be the result of a PRINT PEEK (−16384). If you want to clear the flag that says the keyboard was pressed you have to get a Ø into bit 7 of $CØØØ by putting a Ø into the Clear Keyboard Strobe at $CØ1Ø. In Applesoft, you'd POKE−16368, Ø.

So we compared the accumulator byte (gotten from $CØØØ) with an immediate 128 in 116 and the result of this comparison will set some or none or all of status register flags N, Z or C. C is set when the accumulator is equal to or greater than the data it's compared to. So that's where line 117 comes in: if $CØØØ's "keyboard-was-pressed" bit (#7) is set then the accumulator will certainly be greater than or equal to 128, since even if all the remaining bits in that byte were Ø, the value of the byte with a 1 in it's most significant (8th bit, called #7) bit will be 128.

Line 117 says that if the keyboard flag is set then go to 119 which will return us back to the Applesoft program which is the "driver" (sends input to) for TEST D (CALL 2186), the machine language program that does the animation for C of 28A. The way line 117 says this is with the conditional branch instruction BCS (branch on clear set). This instruction checks C and branches if it's set, which in this case would mean that the CMP in 116 found a number >127 in the accumulator.

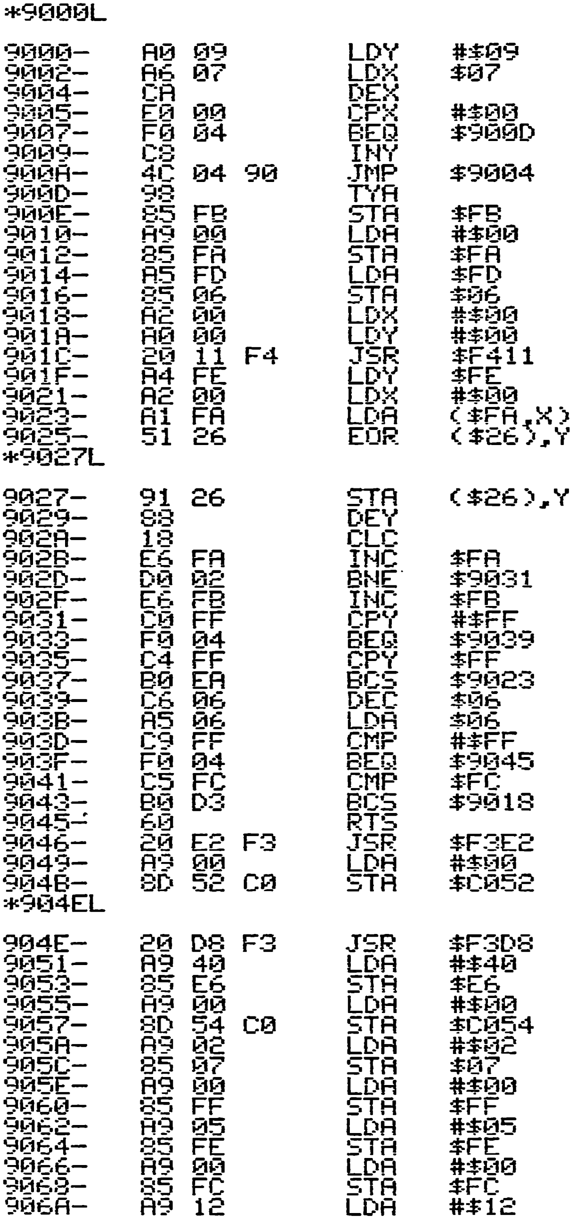
This should have cleared up any questions about what TEST D is about and how it works. It simply draws the shape sequence, which involves 7 draw/erase actions, and then adjusts both the shape # and hor. coordinates, and then checks for either a keyboard hit or a limit-reaching (hor. coord. of HR>35), and then draws the shape sequence again at the adjusted HR and HL and this continues until a hor. limit is reached or a keyboard is hit. If the limit is reached, the next sequence will be back at the leftmost HL and HR in the routine, while if the keyboard has been hit, you'll find yourself back in the Applesoft program being asked that age-old question: "Want to do it again?".

I repeat: your animation programs require no such keyboard-hit detector. Nor do they need the scanner portion of TEST D --- the drawing portion will suffice. Re-assemble accordingly, if you have an assembler. If you don't, it's okay to assemble from the monitor or use the routine as it is.

Now a strange phenomenon will be observed in C of 28A: depending upon the delay-loop inputs you give you'll find that shapes can flicker, strobe, slowly disappear and reappear, etc. It all hinges on the duration each shape is allowed to be visible, which in turn effects the frequency of shapes draw/erase cycles. With a high byte of 2 and a low byte of 249 the effect really becomes very pronounced. The shape is extremely clear and well-animated and non-flickery, even though it's only 1-page animation; however the shape very slowly disappears and reappears. In fact, it sometime starts invisibly (when you first begin the routine), and it sometimes starts partly or wholly visible. This phenomenon relates, it would seem, to certain mysterious timing/frequency factors within the computer and how they mesh/fail to harmonize with the physiology of the human eye, relative to scanning frequencies or "critical flicker frequency", a term from freshman pychology classes. If anyone can pin down the phenomenon better than the above, please let me know. As the saying goes: Don Fudge is NOT Albert Einstein!

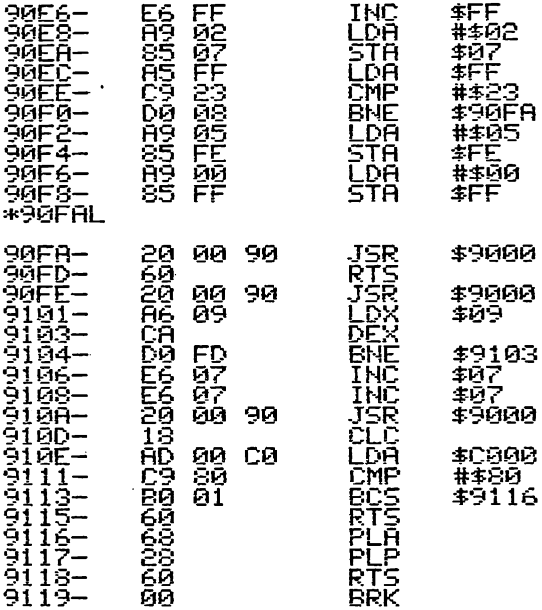
Now let's look at a 2-page flipping animation program. It's B on 28A and the binary file routine is called TEST E (CALL36934). There is no LISA-EXECable source-code file for this binary file, so we'll do a monitor disassembly:

PRINTOUT #17



```
*9000L

9000-   A0 09        LDY   #$09
9002-   A6 07        LDX   $07
9004-   CA           DEX
9005-   E0 00        CPX   #$00
9007-   F0 04        BEQ   $900D
9009-   C8           INY
900A-   4C 04 90     JMP   $9004
900D-   98           TYA
900E-   85 FB        STA   $FB
9010-   A9 00        LDA   #$00
9012-   85 FA        STA   $FA
9014-   A5 FD        LDA   $FD
9016-   85 06        STA   $06
9018-   A2 00        LDX   #$00
901A-   A0 00        LDY   #$00
901C-   20 11 F4     JSR   $F411
901F-   A4 FE        LDY   $FE
9021-   A2 00        LDX   #$00
9023-   A1 FA        LDA   ($FA,X)
9025-   51 26        EOR   ($26),Y
*9027L

9027-   91 26        STA   ($26),Y
9029-   88           DEY
902A-   18           CLC
902B-   E6 FA        INC   $FA
902D-   D0 02        BNE   $9031
902F-   E6 FB        INC   $FB
9031-   C0 FF        CPY   #$FF
9033-   F0 04        BEQ   $9039
9035-   C4 FF        CPY   $FF
9037-   B0 EA        BCS   $9023
9039-   C6 06        DEC   $06
903B-   A5 06        LDA   $06
903D-   C9 FF        CMP   #$FF
903F-   F0 04        BEQ   $9045
9041-   C5 FC        CMP   $FC
9043-   B0 D3        BCS   $9018
9045-   60           RTS
9046-   20 E2 F3     JSR   $F3E2
9049-   A9 00        LDA   #$00
904B-   8D 52 C0     STA   $C052
*904EL

904E-   20 D8 F3     JSR   $F3D8
9051-   A9 40        LDA   #$40
9053-   85 E6        STA   $E6
9055-   A9 00        LDA   #$00
9057-   8D 54 C0     STA   $C054
905A-   A9 02        LDA   #$02
905C-   85 07        STA   $07
905E-   A9 00        LDA   #$00
9060-   85 FF        STA   $FF
9062-   A9 05        LDA   #$05
9064-   85 FE        STA   $FE
9066-   A9 00        LDA   #$00
9068-   85 FC        STA   $FC
906A-   A9 12        LDA   #$12
```

```
906C-   85 FD       STA   $FD
906E-   20 00 90    JSR   $9000
9071-   A9 00       LDA   #$00
9073-   8D 55 C0    STA   $C055
9076-   A9 20       LDA   #$20
9078-   85 E6       STA   $E6
*907AL

907A-   A9 01       LDA   #$01
907C-   85 07       STA   $07
907E-   20 00 90    JSR   $9000
9081-   A9 02       LDA   #$02
9083-   85 07       STA   $07
9085-   A9 20       LDA   #$20
9087-   85 E6       STA   $E6
9089-   A9 00       LDA   #$00
908B-   8D 55 C0    STA   $C055
908E-   20 A0 90    JSR   $90A0
9091-   A9 40       LDA   #$40
9093-   85 E6       STA   $E6
9095-   A9 00       LDA   #$00
9097-   8D 54 C0    STA   $C054
909A-   20 A0 90    JSR   $90A0
909D-   4C 85 90    JMP   $9085
90A0-   C6 07       DEC   $07
90A2-   A5 07       LDA   $07
90A4-   C9 06       CMP   #$06
90A6-   D0 1D       BNE   $90C5
*90A8L

90A8-   20 00 90    JSR   $9000
90AB-   A9 01       LDA   #$01
90AD-   85 07       STA   $07
90AF-   E6 FE       INC   $FE
90B1-   E6 FF       INC   $FF
90B3-   A5 FF       LDA   $FF
90B5-   C9 23       CMP   #$23
90B7-   D0 08       BNE   $90C1
90B9-   A9 05       LDA   #$05
90BB-   85 FE       STA   $FE
90BD-   A9 00       LDA   #$00
90BF-   85 FF       STA   $FF
90C1-   20 00 90    JSR   $9000
90C4-   60          RTS
90C5-   A5 07       LDA   $07
90C7-   C9 00       CMP   #$00
90C9-   D0 33       BNE   $90FE
90CB-   A9 07       LDA   #$07
90CD-   85 07       STA   $07
90CF-   C6 FE       DEC   $FE
*90D1L

90D1-   C6 FF       DEC   $FF
90D3-   A5 FE       LDA   $FE
90D5-   C9 04       CMP   #$04
90D7-   D0 08       BNE   $90E1
90D9-   A9 22       LDA   #$22
90DB-   85 FF       STA   $FF
90DD-   A9 27       LDA   #$27
90DF-   85 FE       STA   $FE
90E1-   20 00 90    JSR   $9000
90E4-   E6 FE       INC   $FE
```



```
90E6-   E6 FF        INC   $FF
90E8-   A9 02        LDA   #$02
90EA-   85 07        STA   $07
90EC-   A5 FF        LDA   $FF
90EE-   C9 23        CMP   #$23
90F0-   D0 08        BNE   $90FA
90F2-   A9 05        LDA   #$05
90F4-   85 FE        STA   $FE
90F6-   A9 00        LDA   #$00
90F8-   85 FF        STA   $FF
*90FAL

90FA-   20 00 90     JSR   $9000
90FD-   60           RTS
90FE-   20 00 90     JSR   $9000
9101-   A6 09        LDX   $09
9103-   CA           DEX
9104-   D0 FD        BNE   $9103
9106-   E6 07        INC   $07
9108-   E6 07        INC   $07
910A-   20 00 90     JSR   $9000
910D-   18           CLC
910E-   AD 00 C0     LDA   $C000
9111-   C9 80        CMP   #$80
9113-   B0 01        BCS   $9116
9115-   60           RTS
9116-   68           PLA
9117-   28           PLP
9118-   60           RTS
9119-   00           BRK
```

Run the program. Two features of its performance stand out immediately: it has no flickery problems or disappearing problems at all, and it is a bit shakier than the previous program.

The non-flickery clearness and hang-in-there-ness are due to the fact that erasing and redrawing happens only when the opposite screen is being displayed. Never is anything displayed that's less than a complete shape.

The shakiness is a bit more complex. It has to do with the fact that when you erase shape 6 and draw shape 1 or when you erase shape 7 and draw shape 2, you need to do many more operations than at any other time in the sequence, and more operations means more time spent doing these operations. The result is a distinct slow-down in the last 2 erase/draw actions in each sequence. Hence the shakiness. But I'll clarify this with the table on the following page.

Which Screen, Which Shape, Draw and Erase Chart

	screen on which to erase/draw	screen displayed	shape # erased	shape # drawn	HL of shape erased	HL of shape drawn
	1	2	1	3	Ø	Ø
	2	1	2	4	Ø	Ø
	1	2	3	5	Ø	Ø
	2	1	4	6	Ø	Ø
	1	2	5	7	Ø	Ø
•	2	1	6	1	Ø	1
••	1	2	7	2	Ø	1
	2	1	1	3	1	1
	1	2	2	4	1	1
	2	1	3	5	1	1
	1	2	4	6	1	1
	2	1	5	7	1	1
•	1	2	6	1	1	2
••	2	1	7	2	1	2
	1	2	1	3	2	2
	↓	↓	↓	↓	↓	↓
	2	1	5	7	34	34
•	1	2	6	1	34	Ø
••	2	1	7	2	34	Ø
	1	2	1	3	Ø	Ø

Above is a table/chart that shows what's happening in TEST E (CALL36934) when B of 28A is run. In column 1 is the screen (page 1 or 2) on which erase/draw operations are being performed. The next column shows the screen currently displayed. The next column shows the shape # of the shape that will be erased. The next column shows the shape # of the shape that will be drawn to replace the one just erased. The next column shows the HL (hor. left coord. byte) of the shape to be erased, while the final column gives the HL of the shape to be drawn.

Notice how sequences are dealt with. Before the events in the chart ever commence, the instructions at $9Ø46 to $9Ø84 are carried out. These instructions initialize both hi-res pages and draw shape #1 on page 1 and shape #2 on page 2, after giving the following coordinates for most of the first sequence: HL=Ø, HR=5, VB=18 ($12), VT=Ø. This gets a different shape on each screen and readys the routine for the erase/draw sequential cycle that starts at $9Ø85.

Once the chart's sequence begins, one screen is displayed while the other screen erases a shape #X and then draws a shape #X+2. Before you get confused, let's take a look at column 4. Remember, shapes 1 and 2 already exist on screens 1 and 2, respectively. If you follow down column 4, you'll see that the net effect of all this is that shape 3 gets drawn on screen 1 and displayed during the next operation; then shape 4 gets drawn on screen 2 while shape 3 is being displayed on screen 1, then shape 5 gets drawn on screen 1 while shape 4 is being displayed on screen 2, etc. I've left out the part about erasing, for now. So forget erasing until you can follow through both this paragraph and the chart and know exactly what's going on.

If you're in this paragraph, it should be because you totally understand all that's come before it. If you don't, then BEWARE ALL YE WHO ENTER HERE. Look at column 3. Notice how the shape #'s are always 2 behind the next column's shape #s. In other words, just before shape #3 is drawn, shape #1 is erased, and just before #4 is drawn, #2 is erased, etc. You see, what we're trying to do is flip back and forth between the hi-res pages for each of the 7 parts of each sequence so that no flickeriness results from display of screens being erased/drawn upon. Draw 1 on 1, draw 2 on 2, draw 3 on 1, draw 4 on 2, draw 5 on 1, etc. However, just drawing the correct shape is simply not sufficient --- we must also erase the old shape on the screen just before drawing a new one 2 shape #s ahead. That's what column 3 is about. It's a simple enough concept: draw 1 on 1, draw 2 on 2, replace 1 with 3 on 1, replace 2 with 4 on 2, replace 3 with 5 on 1, replace 4 with 6 on 2, replace 5 with 7 on 1, replace 6 with --- oops!

Ran out of shape #s! So now we need to not only replace 6 with 1 but also move HL and HR over one before drawing 1, since we'll be on the next sequence when we draw the new shape #1.

Now, for the next shape we need to replace 7 with 2 and move HL and HR back one for the erasing of 7, and then back to the new hor. coords. for the drawing of #2. It would be wise to study the chart until everything seems perfectly clear and logical. Remember, whenever you get to the end of one sequence you must move your horizontal coordinates HL and HR over one (increment them). And when the situation is like it is on the chart, you may have to increment, decrement, increment, rather than just a simple increment like the way it is with one-page animation. In the chart you'll stay with the old HL and HR until you get to erasing 6 and drawing 1. Leave old ones for erasing 6, increment before drawing 1, and then in the next erase/draw cycle decrement before erasing 7 and increment (HL and HR) again before drawing 2. Leave the hor. coords. alone until you reach 6 and 1 again.

As you can see, 2-page animation is a bit involved, compared to 1-page animation, especially when you start creating assembly programs to do everything that happens on the chart. The results are worth the effort, however, as you'll see pretty quick. Don't try to understand TEST E (CALL36934) from the disassembled list-out. At least not until you've studied this entire book and all the listings included. You might be ready to try it by then. You may wish to have a few things pointed out to you, however:

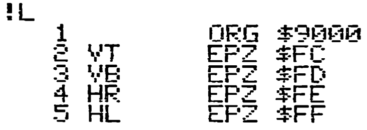
From $9085 to $909F is the main page-switching loop. From $9000 to $9045 is the usual drawing routine. From $90A0 to the end of the program is a subroutine gotten to by JSR $90A0 which occurs in the main page-switching loop. This subroutine does everything **except** switch pages for display and switch pages for erase/draw actions. At $910D is the keyboard-press checking routine, which goes to the end of the program. AT $9116 and $9117 are commands that have the net effect of a POP in BASIC: since JSR $90A0 is routine JSRed to after the program has already CALLed 36934 to get us into the mach. lang. routine in the first place, it stands to reason that a simple RTS will not suffice to get us back into the BASIC program. We'll have to pull 2 addresses from the stack in our RTS --- the first one will only be telling us to go back to the main page-switching loop. The remainder of the keyboard-press routine has already been explained --- and remember, you aren't likey to actually need such a routine in a game! You may need a routine to determine keyboard input, but it isn't terribly likely to just send you back to a BASIC program, which is all this one does.

Other addresses of note: at $90A4 we find out if we're at shape #6 yet (one asterisk on the chart) so that we can perform the special functions needed at this point by going to $90A8 and erasing 6, then incrementing hor. coords., then checking for being too close to the right edge of the screen, then drawing 1. Another address: $90C5 --- here we check to see if the shape # is Ø(when the shape # went from 6 to 1 the time before it left 1 in $7, so now since subroutine $9ØAØ starts with DEC $7, 1 will change to Ø just before $9ØC5 checks it). If it is Ø then we're at the 2nd special part (2 asterisks on the chart) of the subroutine. Here we are sent to $9ØCB and put shape #7 in $7, and then HL and HR are decremented and then we're sent to $9ØE1 to erase old 7 and then HL and HR are incremented and a new #2 is drawn at these newer coordinates. Another address is $9ØD3, where we load the accumulator with $FE (HR) and check to see if it's a 4 ($FE was just decremented 2 lines above this otherwise it'd be a 5 we'd check for) which would mean we're at the 2nd to last line of the chart. This would also mean that during the last erase/draw $FF (HL) was found to be 35 ($23) at address $9ØB7, so it loaded 5 into HR and Ø into HL from $9ØB9-$9ØCØ. This means that before HL equalled 35 it was 34 and still "okay", but when it was found to equal its limit in $9ØB5-$9ØB8, it got sent back to the left-most hor. coords. along with its partner, HR. All of this is merely the way a shape sequence gets its hor. coordinates moved back to its starting coordinates once it exceeds its allowable limits. This may or may not be necessary in the games you write.

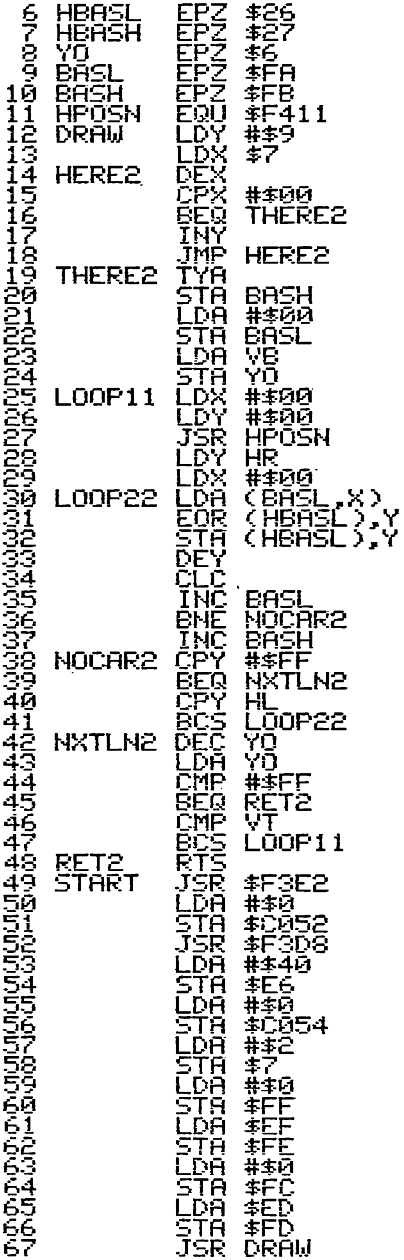
So let's hope you've got the basic idea of 2-page animation. Don't sweat the details yet --- a disassembled list-out is a poor way to learn details, unless you're a real assembly wizard. I suggest studying the details in the next list-out more carefully.

Now let's look at a 2-page flipping program with inputs, D of 28A. In this program you get to choose what shape table sequence you want, the starting shape # for the sequence (up to 17), the width and height of your shapes, the step-size, the right boundary for the left side of your shape if you're moving right-wards, the "high" and "low" delay loop bytes, and which direction you wish to travel --- left or right. The binary files that get loaded for D of 28A are either TEST F (CALL 36934), for left-to-right movement, or TEST G (CALL 36934), for right-to-left moving. The source-code files for these programs are called TEST F and TEST G. Here is TEST F:

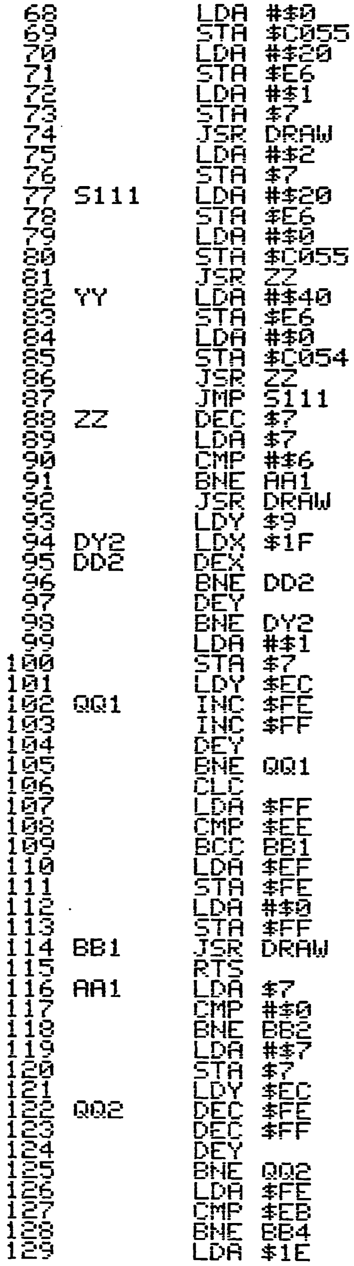
PRINTOUT #18



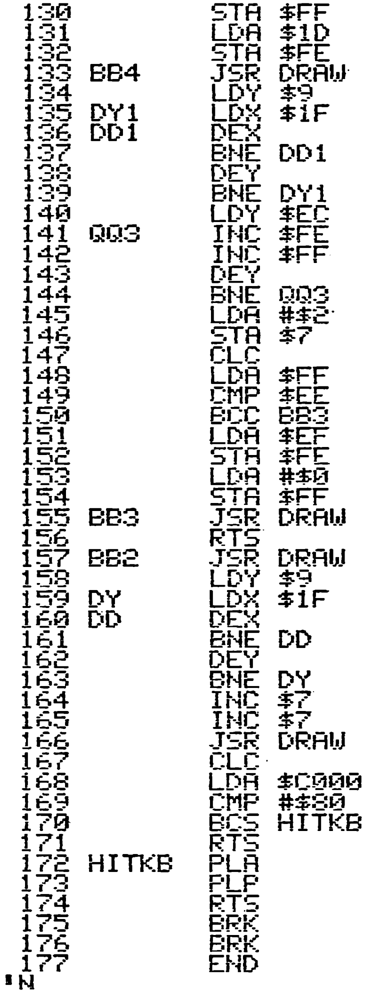
```
!L
     1            ORG $9ØØØ
     2 VT         EPZ $FC
     3 VB         EPZ $FD
     4 HR         EPZ $FE
     5 HL         EPZ $FF
```



```
HBASL   EPZ $26
HBASH   EPZ $27
YO      EPZ $6
BASL    EPZ $FA
BASH    EPZ $FB
HPOSN   EQU $F411
DRAW    LDY #$9
        LDX $7
HERE2   DEX
        CPX #$00
        BEQ THERE2
        INY
        JMP HERE2
THERE2  TYA
        STA BASH
        LDA #$00
        STA BASL
        LDA VB
        STA YO
LOOP11  LDX #$00
        LDY #$00
        JSR HPOSN
        LDY HR
        LDX #$00
LOOP22  LDA (BASL,X)
        EOR (HBASL),Y
        STA (HBASL),Y
        DEY
        CLC
        INC BASL
        BNE NOCAR2
        INC BASH
NOCAR2  CPY #$FF
        BEQ NXTLN2
        CPY HL
        BCS LOOP22
NXTLN2  DEC YO
        LDA YO
        CMP #$FF
        BEQ RET2
        CMP VT
        BCS LOOP11
RET2    RTS
START   JSR $F3E2
        LDA #$0
        STA $C052
        JSR $F3D8
        LDA #$40
        STA $E6
        LDA #$0
        STA $C054
        LDA #$2
        STA $7
        LDA #$0
        STA $FF
        LDA $EF
        STA $FE
        LDA #$0
        STA $FC
        LDA $ED
        STA $FD
        JSR DRAW
```



```
 68           LDA  #$0
 69           STA  $C055
 70           LDA  #$20
 71           STA  $E6
 72           LDA  #$1
 73           STA  $7
 74           JSR  DRAW
 75           LDA  #$2
 76           STA  $7
 77  S111     LDA  #$20
 78           STA  $E6
 79           LDA  #$0
 80           STA  $C055
 81           JSR  ZZ
 82  YY       LDA  #$40
 83           STA  $E6
 84           LDA  #$0
 85           STA  $C054
 86           JSR  ZZ
 87           JMP  S111
 88  ZZ       DEC  $7
 89           LDA  $7
 90           CMP  #$6
 91           BNE  AA1
 92           JSR  DRAW
 93           LDY  $9
 94  DY2      LDX  $1F
 95  DD2      DEX
 96           BNE  DD2
 97           DEY
 98           BNE  DY2
 99           LDA  #$1
100           STA  $7
101           LDY  $EC
102  QQ1      INC  $FE
103           INC  $FF
104           DEY
105           BNE  QQ1
106           CLC
107           LDA  $FF
108           CMP  $EE
109           BCC  BB1
110           LDA  $EF
111           STA  $FE
112           LDA  #$0
113           STA  $FF
114  BB1      JSR  DRAW
115           RTS
116  AA1      LDA  $7
117           CMP  #$0
118           BNE  BB2
119           LDA  #$7
120           STA  $7
121           LDY  $EC
122  QQ2      DEC  $FE
123           DEC  $FF
124           DEY
125           BNE  QQ2
126           LDA  $FE
127           CMP  $EB
128           BNE  BB4
129           LDA  $1E
```



```
130               STA  $FF
131               LDA  $1D
132               STA  $FE
133  BB4          JSR  DRAW
134               LDY  $9
135  DY1          LDX  $1F
136  DD1          DEX
137               BNE  DD1
138               DEY
139               BNE  DY1
140               LDY  $EC
141  QQ3          INC  $FE
142               INC  $FF
143               DEY
144               BNE  QQ3
145               LDA  #$2
146               STA  $7
147               CLC
148               LDA  $FF
149               CMP  $EE
150               BCC  BB3
151               LDA  $EF
152               STA  $FE
153               LDA  #$0
154               STA  $FF
155  BB3          JSR  DRAW
156               RTS
157  BB2          JSR  DRAW
158               LDY  $9
159  DY           LDX  $1F
160  DD           DEX
161               BNE  DD
162               DEY
163               BNE  DY
164               INC  $7
165               INC  $7
166               JSR  DRAW
167               CLC
168               LDA  $C000
169               CMP  #$80
170               BCS  HITKB
171               RTS
172  HITKB        PLA
173               PLP
174               RTS
175               BRK
176               BRK
177               END
▮N
```

Up to line 48 there's nothing but the drawing routine. And from lines 167 to 177 is the keyboard-hitting routine already covered. From 93-98 and from 134-139 and from 158-163 are standard "high" and "low" delay loops where the delay is one number multiplied by the other number, essentially. Note that the "high" byte is stored in $9 but the "low" byte is stored in $1F.

From **lines 49-76** is the "get ready" section. Here's what happens there:

Line 49 is the HGR command.

Line 50-51 make HGR full-screen graphics.

Line 52 is the HGR2 command.

Line 53-54 says do it to page 2 (draw on it).

Line 55-56 says display page 1.

Line 57-58 loads 2 into $7, the shape # holder.

Line 59-60 loads a Øinto HL, which is $FF.

Line 61-62 loads the width from the width-holder address ($EF) into HR, which is $FE.

Line 63-64 loads a Ø into VT, which is $FC.

Line 65-66 loads the height from the height-holder address ($ED) into VB, which is $FD.

Line 67 draws shape #2 on page 2.

Line 68-69 displays page 2.

Line 70-71 says draw on page 1.

Line 72-73 says store a 1 in $7, the shape # holder.

Line 74 draws shape #1 on page 1.

Line 75-76 reloads $7, the shape # holder, with a 2.

So now let's see what happens in the main page-switching loop, from line 77 to line 87.

Line 77-78 says draw on page 1.

Line 79-80 says display page 2.

Line 81 says go to drawing subroutine at line 88 but don't forget to return again (a JSR is like a GOSUB in BASIC).

Line 82-83 says draw on page 2.

Line 84-85 says display page 1.

Line 86 says go again to drawing routine at line 88.

Line 87 says jump, without returning, to the beginning of the main page-switching loop again.

Now let's look at the drawing subroutine at line 88. The line label is ZZ, which gives us something easy to call it in lines 81 and 86. There's no other real significance to line labels. Make them so they help you find where things are, if you're assembling.

Line 88 temporarily decrements $7 so you'll be erasing the correct shape. Consult the chart again --- the 1st line says erase 1, draw 3 and the 2nd line says erase 2, draw 4. The 3 in the 1st line had to be changed into the 2 in the 2nd line, so $7 was decremented temporarily here.

Line 89-90 checks to see if you're at shape #6 yet.

Line 91 branches way down to line 116 if you're not at shape #6 yet.

Line 92 --- your shape # is 6 if you didn't branch so this line erases your old shape.

Line 93-98 delay loop.

Line 99-100 loads $7 with 1 since in erase/draw cycles where you erase #6, you also draw #1.

Line 101 loads the step-size into the Y register; we'll be storing the step-size in $EC.

Line102-103 increments HL($FF) and HR($FE) since next we'll be starting the sequence over with shape #1.

Line 104 decrements Y, where the step-size was put.

Line 105 branches back up to line 1Ø2 if Y has not yet been reduced to Ø. In effect you go through this loop of incrementing HL and HR the number of times that is equal to the step-size.

Line 106 clears the carry flag because in line 109 we'll be branching according to the status of this flag, so it must not start out set.

Line 107-108 puts the current HL into the accumulator where it gets compared with $EE, in which address we'll be storing the right boundary of HL, as decided by inputs previous to this routine.

Line 109 if the carry is clear, which means the accumulator (with current HL value) was less than the # in $EE (the HL right boundary) then we move on down to line 114, draw the shape, and return.

Line 110-111 if we have reached or exceeded the right boundary, then we end up here, and load the block-shape's width into HR, after getting this # from $EF.

Line 112-113 if we reached or exceeded the boundary, as above, then the HL becomes 0 again.

Line 114-115 is for drawing the shape you're on and returning to the main page-switching loop from whence you came.

Line 116-117 finds out if the shape #, which couldn't have been less than 1 upon entering this drawing subroutine at line 88, was 1 but quickly changed to 0 in line 88.

Line 118 sends you all the way to 157 if the # in $7 <>0.

Line 119-120 if you're here then you're in the situation represented by 2 asterisks on the chart. First you load a 7 into $7.

Line 121-125 is the same as line 101-105.

Line 126-127 puts HR into the accumulator and compares it to a number equal to the width minus the step-size. This pre-determined number will be stored at $EB.

Line 128 if HR isn't equal to width minus step-size, then you'll be sent to line 133. What's happening here relates to the 2nd to last line on our chart. If the last erase/draw cycle reached the boundary stored in $EE and started the shape over at HL=0 and HR=width, then HR will now equal width minus step-size, due to lines 121-125 subtracting the step-size from the current HR and HL.

Line 129-130 if HR is equal to width minus step-size, then you'll end up here getting a number from $1E that's predetermined and equal to the right boundary of HL minus the step size. This will be your new HL. All this is to correctly calculate where the right-most shape was when it was drawn so that you may now successfully erase it. The "minus step-size" part of this is to make up for what will soon be happening in lines 140-144, where step-size will be added to HL and HR.

Line 131-132 is finishing the job started by lines 129-130. Your new HR will be the same number as that in $1E, except that it will have been stored in $1D and it will be $1E plus the width. Or think of this formula for $1D: (right boundary for HL minus step-size) plus width.

Line 133 erases the shape whose # is 7.

Line 134-139 delay loop.

Line 140-144 is the same as line 101-105.

Line 145-146 puts a 2 in $7, the shape # holder. See the 2nd to the last line of the chart, column 3.

Line 147 clears the carry so line 150 will work right.

Line 148-149 compares HL to the right boundary for HL, which is stored in $EE.

Line 150 sends you to line 155 if the present HL is not as large as the right boundary for HL, where you'll draw #2 and return.

Line 151-152 If you're here, then it means your situation is the 2nd to the last line of the chart and you need to decrement HL and HR to their lowest (left-most) values before drawing #2. Width gets stored in HR ($FE).

Line 153-154 finishes the job started by lines 151-152 ---decrements HL down to 0. Only now is it okay to continue to line 155 and draw #2.

Line 155-156 draws #2 and returns.

Line 157 begins the subroutine that you end up at if you're at neither of the types of places that got asterisks on the chart. In other words, you weren't ready to erase 6 and draw 1 or erase 7 and draw 2 if you made it down to here. This line erases your shape of #1 to #5.

Line 158-163 delay loop.

Line 164-165 increments $7 twice so that the shape it will replace the one it just erased with will be 2 numbers greater than the erased one.

Line 166 draws shape 3-7 (one of them).

Line 167 clears carry so line 170 will work right.

Line 168-169 checks keyboard-press flag in $C000.

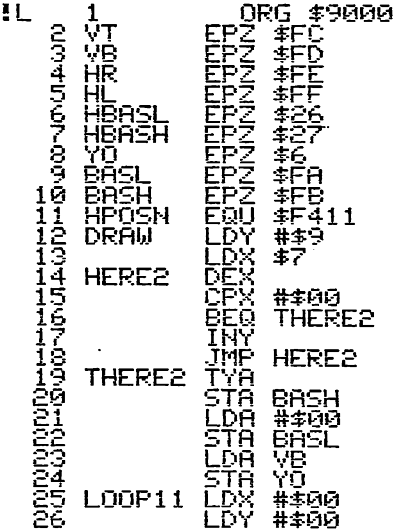
Line 170 sends you to 172 if key was pressed.

Line 171 sends you back to main page-switching loops.

Line 172-174 POPS stacks and returns you to BASIC program due to keyboard press.

Now let's look at TEST G, which moves you towards the left instead of the right like TEST F does. We'll only be looking at ways G and F differ. If something seems skipped-over or unexplained, that's because it's **already** been explained in this chapter:

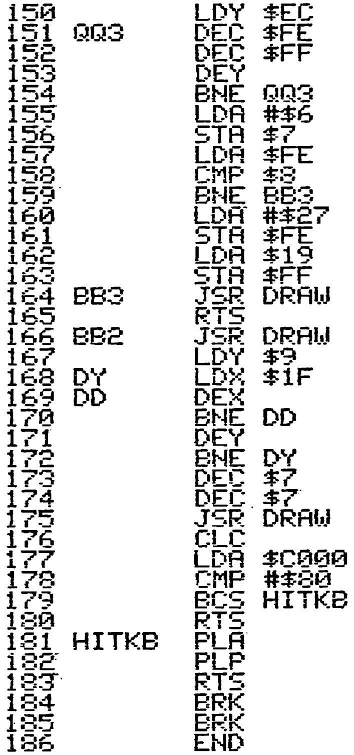
PRINTOUT #19



```
!L   1          ORG $9000
     2 VT       EPZ $FC
     3 VB       EPZ $FD
     4 HR       EPZ $FE
     5 HL       EPZ $FF
     6 HBASL    EPZ $26
     7 HBASH    EPZ $27
     8 YO       EPZ $6
     9 BASL     EPZ $FA
    10 BASH     EPZ $FB
    11 HPOSN    EQU $F411
    12 DRAW     LDY #$9
    13          LDX $7
    14 HERE2    DEX
    15          CPX #$00
    16          BEQ THERE2
    17          INY
    18          JMP HERE2
    19 THERE2   TYA
    20          STA BASH
    21          LDA #$00
    22          STA BASL
    23          LDA VB
    24          STA YO
    25 LOOP11   LDX #$00
    26          LDY #$00
```

```
27          JSR HPOSN
28          LDY HR
29          LDX #$00
30 LOOP22   LDA (BASL,X)
31          EOR (HBASL),Y
32          STA (HBASL),Y
33          DEY
34          CLC
35          INC BASL
36          BNE NOCAR2
37          INC BASH
38 NOCAR2   CPY #$FF
39          BEQ NXTLN2
40          CPY HL
41          BCS LOOP22
42 NXTLN2   DEC YO
43          LDA YO
44          CMP #$FF
45          BEQ RET2
46          CMP VT
47          BCS LOOP11
48 RET2     RTS
49 START    JSR $F3E2
50          LDA #$0
51          STA $C052
52          JSR $F3D8
53          LDA #$40
54          STA $E6
55          LDA #$0
56          STA $C054
57          LDA #$6
58          STA $7
59          LDA $19
60          STA $FF
61          LDA #$27
62          STA $FE
63          LDA #$0
64          STA $FC
65          LDA $ED
66          STA $FD
67          JSR DRAW
68          LDA #$0
69          STA $C055
70          LDA #$20
71          STA $E6
72          LDA #$7
73          STA $7
74          JSR DRAW
75          LDA #$6
76          STA $7
77 S111     LDA #$20
78          STA $E6
79          LDA #$0
80          STA $C055
81          JSR ZZ
82 YY       LDA #$40
83          STA $E6
84          LDA #$0
85          STA $C054
86          JSR ZZ
87          JMP S111
```

```
 88 ZZ      INC $7
 89         LDA $7
 90         CMP #$2
 91         BNE AA1
 92         JSR DRAW
 93         LDY $9
 94 DY2     LDX $1F
 95 DD2     DEX
 96         BNE DD2
 97         DEY
 98         BNE DY2
 99         LDA #$7
100         STA $7
101         LDY $EC
102 QQ1     DEC $FE
103         DEC $FF
104         DEY
105         BNE QQ1
106         LDA $FE
107         CMP $EF
108         BCS BB1
109         CLC
110         LDA $FE
111         ADC $EC
112         STA $1D
113         LDA $FF
114         ADC $EC
115         STA $1E
116         LDA #$27
117         STA $FE
118         LDA $19
119         STA $FF
120 BB1     JSR DRAW
121         RTS
122 AA1     LDA $7
123         CMP #$8
124         BNE BB2
125         LDA #$1
126         STA $7
127         LDY $EC
128 QQ2     INC $FE
129         INC $FF
130         DEY
131         BNE QQ2
132         LDA $FE
133         CMP $EB
134         BNE BB4
135         SEC
136         LDA $1D
137         SBC $EC
138         STA $8
139         LDA $1E
140         STA $FF
141         LDA $1D
142         STA $FE
143 BB4     JSR DRAW
144         LDY $9
145 DY1     LDX $1F
146 DD1     DEX
147         BNE DD1
148         DEY
149         BNE DY1
```



```
150          LDY $EC
151 QQ3      DEC $FE
152          DEC $FF
153          DEY
154          BNE QQ3
155          LDA #$6
156          STA $7
157          LDA $FE
158          CMP $8
159          BNE BB3
160          LDA #$27
161          STA $FE
162          LDA $19
163          STA $FF
164 BB3      JSR DRAW
165          RTS
166 BB2      JSR DRAW
167          LDY $9
168 DY       LDX $1F
169 DD       DEX
170          BNE DD
171          DEY
172          BNE DY
173          DEC $7
174          DEC $7
175          JSR DRAW
176          CLC
177          LDA $C000
178          CMP #$80
179          BCS HITKB
180          RTS
181 HITKB    PLA
182          PLP
183          RTS
184          BRK
185          BRK
186          END
```

The lines that differ between TEST F and TEST G are as follows:

Line 57-58 puts a 6 into $7 rather than a 2 --- we must start at/near the end of the sequence and work backwards since the shape movements are to be left-wards.

Line 59-60 start HL towards the right for the same reason as the above. $19 is where this right-most coord. for HL is stored --- its formula is 39 minus the width.

Line 61-62 puts a 39 into HR --- 39 is the # of the right-most possible byte on the hi-res screen.

Line 72-73 puts a 7 in $7 --- the sequence must run backwards.

Line 75-76 puts a 6 in $7 --- same reason.

Line 88 is INC $7 rather than DEC $7 as in TEST F --- all decrementing and incrementing between these 2 programs will be opposite of each other because they're moving in opposite directions.

Line 90 here we're looking for shape #2 to be analogous to the 1-asterisk lines in the chart where 6 will be replaced by 1. We're looking for 2 to be replaced by 7.

Line 99-100 as stated above, we want 2 to be replaced by 7.

Line 106-107 we find out if HR ($FE) has gotten to the # that is also the width (stored in $EF) which would also mean that HL was at 0.

Line 108 if HR > width then you get sent to line 120.

Line 110-115 if HR isn't > width then step-size ($EC) is added to HR and stored in $1D and step size ($EC) is added to HL and stored in $1E. These #s will be used later.

Line 116-119 since the left-most hor. coords. have been reached, we now must put 39 into HR and 39-width in HL in order to start the screen-crossing again.

Line 122-123 we find out if 8 (it would be 7 except for INC $7 at line 88) is in $7, the shape # holder. This is analogous to the 2-asterisk situations on the chart where 7 gets erased and 2 gets drawn in its place.

Line 125-126 1 gets erased (and replaced by 6 later).

Line 135 carry is set, so it will be ready for subtraction at line 137 where a "carry" might be needed.

Line 136-138 subtracts step size from the # we saved there ($1D) and stores this new # in $8 to use to compare with HR later.

Line 139-142 situation comparable to the 2-asterisk situation of the 2nd line from the bottom in the chart we've been referring to. We now need to erase the "34" (which in our situation would be a low number dependent upon shape width), so our version of "34" has been calculated and put into $1D, which is now loaded into HR, and a number stored in $1E that is one width smaller than $1D has been calculated and is now loaded into HL.

Line 143 the 2-asterisk **erasing** situation now runs the drawing subroutine and erases the shape (#1) in the previous sequence's hor. coord. position.

Line 155-156 shape # in $7 changed to 6 for drawing 6.

Line 157-158 determines if HR is the **left-most allowable HR** previously calculated and stuck in $8. If it is, it means that this drawing of #6 is analogous to the 2nd line from the bottom on the chart and we need to jump our HR and HL across the screen to begin a new screen-crossing beginning at HL=39-width and HR=39. (Notice that the **other** hi-res page has already begun its screen-crossing in the 3rd to last line of the chart where 34 has changed to 0. Now we need to make **this** hi-res screen follow suit.

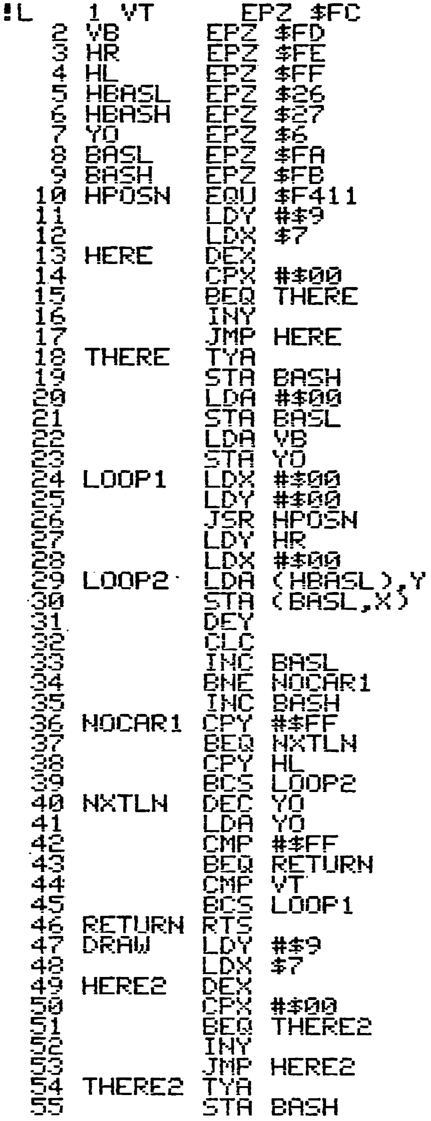
Line 159 if a 2-asterisk (2nd line from the bottom) chart situation was not found to be present, you're sent to line 164 to draw and return.

Line 160-163 if a 2nd line-from-the-bottom-of-the-chart situation **was** found, then 39 gets loaded into HR and 39 minus width gets loaded into HL to prepare for drawing #6 in the right place.

The above source-code files create mach. lang. routines TEST F (CALL 36934) and TEST G (CALL 36934). The source-code test files are named TEST F and TEST G. F goes rightwards and G goes leftwards. The routines are a bit different --- there are many ways to handle page-flipping block-shape sequences. Those were 2 ways. Disk 28C has all binary files from TEST A (CALL2186) to TEST S (CALL2125) and all of the source-code files with the exception of TEST E. Also included is the TEST TB! source-code file for TEST TB, and YTABLE, for animation speed improvement. Disk 28C and disk 28D are both unlocked --- you'll have no problem using any of these files, games, sounds, or other routines.

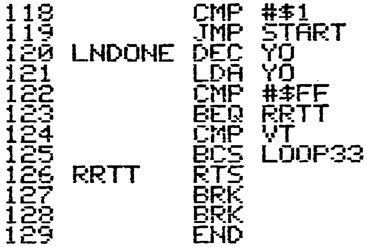
Now let's look at 1 on 28B, the Automatic Block-Shape Creator ---Starting With Block-Shapes. It utilizes a binary file called TEST H (CALL 2186) and the related source-code file is TEST H. Here it is:

PRINTOUT #20



```
!L    1 VT      EPZ $FC
      2 VB      EPZ $FD
      3 HR      EPZ $FE
      4 HL      EPZ $FF
      5 HBASL   EPZ $26
      6 HBASH   EPZ $27
      7 YO      EPZ $6
      8 BASL    EPZ $FA
      9 BASH    EPZ $FB
     10 HPOSN   EQU $F411
     11         LDY #$9
     12         LDX $7
     13 HERE    DEX
     14         CPX #$00
     15         BEQ THERE
     16         INY
     17         JMP HERE
     18 THERE   TYA
     19         STA BASH
     20         LDA #$00
     21         STA BASL
     22         LDA VB
     23         STA YO
     24 LOOP1   LDX #$00
     25         LDY #$00
     26         JSR HPOSN
     27         LDY HR
     28         LDX #$00
     29 LOOP2   LDA (HBASL),Y
     30         STA (BASL,X)
     31         DEY
     32         CLC
     33         INC BASL
     34         BNE NOCAR1
     35         INC BASH
     36 NOCAR1  CPY #$FF
     37         BEQ NXTLN
     38         CPY HL
     39         BCS LOOP2
     40 NXTLN   DEC YO
     41         LDA YO
     42         CMP #$FF
     43         BEQ RETURN
     44         CMP VT
     45         BCS LOOP1
     46 RETURN  RTS
     47 DRAW    LDY #$9
     48         LDX $7
     49 HERE2   DEX
     50         CPX #$00
     51         BEQ THERE2
     52         INY
     53         JMP HERE2
     54 THERE2  TYA
     55         STA BASH
```

```
         LDA #$00
         STA BASL
         LDA VB
         STA YO
LOOP11   LDX #$00
         LDY #$00
         JSR HPOSN
         LDY HR
         LDX #$00
LOOP22   LDA (BASL,X)
         EOR (HBASL),Y
         STA (HBASL),Y
         DEY
         CLC
         INC BASL
         BNE NOCAR2
         INC BASH
NOCAR2   CPY #$FF
         BEQ NXTLN2
         CPY HL
         BCS LOOP22
NXTLN2   DEC YO
         LDA YO
         CMP #$FF
         BEQ RET2
         CMP VT
         BCS LOOP11
RET2     RTS
         LDA VB
         STA YO
LOOP33   LDX #$0
         LDY #$0
         JSR HPOSN
         CLC
         LDY HL
START    LDA #$0
         STA $8
         STA $CF
SHFT     LDA (HBASL),Y
         ROL
         STA (HBASL),Y
         BCS SUB1
         BCC CONT1
SUB1     INC $8
CONT1    CMP #$80
         BCS SET64
         BCC CONT2
SET64    INC $CF
CONT2    LDA $8
         BNE SUB2
         LDA (HBASL),Y
         AND #$7F
         STA (HBASL),Y
         JMP SUB3
SUB2     LDA (HBASL),Y
         ORA #$80
         STA (HBASL),Y
SUB3     CPY HR
         BEQ LNDONE
         INY
         CLC
         LDA $CF
```



```
118          CMP #$1
119          JMP START
12Ø LNDONE   DEC YO
121          LDA YO
122          CMP #$FF
123          BEQ RRTT
124          CMP VT
125          BCS LOOP33
126 RRTT     RTS
127          BRK
128          BRK
129          END
```

From lines 1 to 46 is the scanner, and from lines 47 to 83 is the drawing subroutine. Let's look at the remainder of the list-out:

Line 84-85 saves the VB in YØ ($6).

Line 86-87 gets the X and Y registers ready for the HPØSN routine, which must be entered with hor. low byte in X, hor. high byte in Y, and vertical hi-res coordinate in the accumulator. This last operation was performed in line 84. HPØSN calculated the left-most base address of the vertical line represented by the # put into the accumulator from VB, the vertical coordinate of the bottom of the block-shape.

Line 89 clears the carry since the carry flag will be central to later branching instructions.

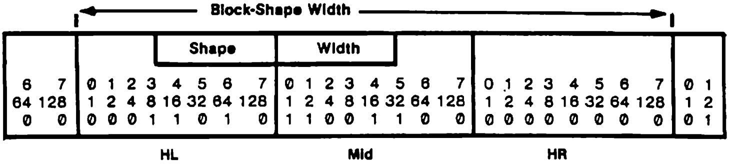
Line 9Ø loads Y with HL --- we'll be dealing with each byte in this shape, one at a time, starting from the lower right-hand corner and working left until at HL, and then moving up a line (to VB-1) and doing it again --- this repeats until the VT line has been completed.

Completed in what way? --- you may well ask. Well, this is an automatic sequence-creator, and, unlike A of 28A it needs no vector shapes to operate. Block-shapes are all it needs --- in fact it only needs **one.** It will make the other 6 in the 7-shape sequence.

So how does it do it? --- you may well ask, since I've stated before that a block's hor. coords. are bytes, and a byte is 7 dots wide on the high-resolution screen (the 8th bit, #7, is not displayed --- it's the color bit). So you can draw a block-shape at HL=5, HR=8, VB=20, VT=1 or at HL=6, HR=9, VB=20, VT=1, but **not** at HL=5 1/7, HR=8 1/7, VB=20, VT=1. Or can't you? There **is** a way! Look in your assembly books and check out these 6502 instructions: ROR and ROL.

At first they sound like the perfect solution to our problem. But then you start realizing that the color bit rotating along with everything else will tend to mess up the works royally. Upon closer inspection you also begin to see that even though the carry bit rotates nicely into bit Ø and bit 7 rotates sweet-as-you-please into the carry bit, this doesn't make a very convenient way of shifting sideways from one byte into the next, since throwing the Ø bit of one byte sideways into the 7 bit of another byte merely gives that 7 bit an irrelevant setting, since it's the #6 bit we want that Ø bit to hop into. Putting it into 7 is a bit silly (or a silly bit, depending upon your outlook).

Let's look at a diagram:

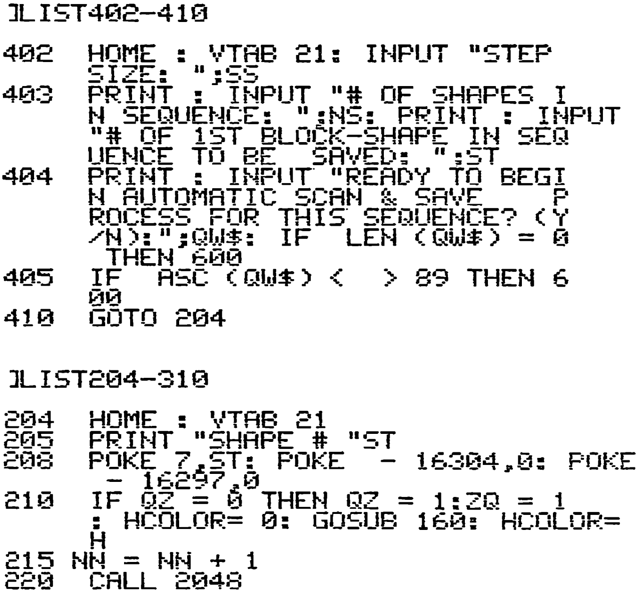


6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1
64 128	1 2 4 8 16 32 64 128	1 2 4 8 16 32 64 128	1 2 4 8 16 32 64 128	1 2
0 0	0 0 0 1 1 0 1 0	1 1 0 0 1 1 0 0	0 0 0 0 0 0 0 0	0 1
	HL	Mid	HR	

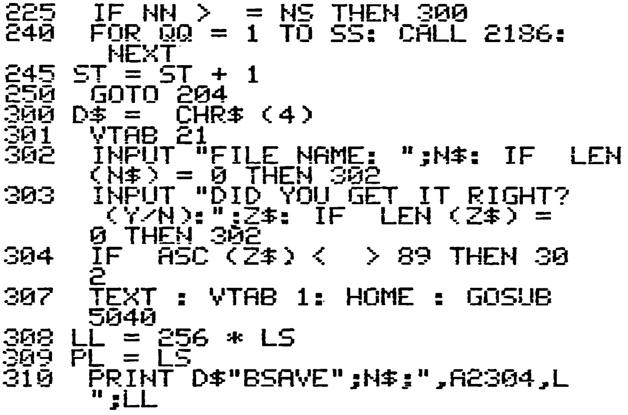
- actual width = 3
- width to give = 2
- bit width of shape within block = 11
- bit width of block-shape, including block (actual) = 21
- extra room to the right is **shifting space**

If we're trying to shift all bits towards the right side of the screen, we need to remember the following: A byte's bits go from right to left, getting bigger. #7 bit is on the left and has a decimal value of 128, while #0 is on the right and has a value of 1. However, when the high-resolution screen displays hi-res bytes, it does it **backwards,** so that the least significant bits are to the left and the most significant bits are to the right. This means that an ROR command will shift bits right, but on the screen it will look like a leftwards shift; ROL shifting will appear rightwards. Here are the parts of 1 of 28B that create and save a block-shape sequence from one block-shape:

PRINTOUT #21



```
]LIST402-410

402  HOME : VTAB 21: INPUT "STEP
     SIZE: ";SS
403  PRINT : INPUT "# OF SHAPES I
     N SEQUENCE: ";NS: PRINT : INPUT
     "# OF 1ST BLOCK-SHAPE IN SEQ
     UENCE TO BE  SAVED: ";ST
404  PRINT : INPUT "READY TO BEGI
     N AUTOMATIC SCAN & SAVE    P
     ROCESS FOR THIS SEQUENCE? (Y
     /N):";QW$: IF  LEN (QW$) = 0
      THEN 600
405  IF  ASC (QW$) <  > 89 THEN 6
     00
410  GOTO 204

]LIST204-310

204  HOME : VTAB 21
205  PRINT "SHAPE # "ST
208  POKE 7,ST: POKE  - 16304,0: POKE
      - 16297,0
210  IF QZ = 0 THEN QZ = 1:ZQ = 1
     : HCOLOR= 0: GOSUB 160: HCOLOR=
     H
215 NN = NN + 1
220  CALL 2048
```



```
225  IF NN >  = NS THEN 300
240  FOR QQ = 1 TO SS: CALL 2186:
      NEXT
245 ST = ST + 1
250  GOTO 204
300 D$ =  CHR$ (4)
301  VTAB 21
302  INPUT "FILE NAME: ";N$: IF  LEN
     (N$) = 0 THEN 302
303  INPUT "DID YOU GET IT RIGHT?
      (Y/N):";Z$: IF  LEN (Z$) =
     0 THEN 302
304  IF  ASC (Z$) <  > 89 THEN 30
     2
307  TEXT : VTAB 1: HOME : GOSUB
     5040
308 LL = 256 * LS
309 PL = LS
310  PRINT D$"BSAVE";N$;",A2304,L
     ";LL
```

The lines from 402 to 410 prompt the necessary inputs from the user: step-size (bit-wise), # of shapes in sequence, first shape # in sequence. The program takes it for granted you'll know enough to load in a block shape and thereby have at least one shape at $900, if not more, at $A00, $B00, etc.

The lines from 204-205 tell what shape # the program is on, although it takes less than a second to create a sequence, so lots o' luck reading the #s that get displayed.

Line 208 puts the first shape's # into $7 and manipulates screen switches to graphics.

Line 210 erases the rectangle you will have drawn around the paddle-defined block-shape you're using for the sequence creation.

Line 215 keeps track of how many shapes get drawn.

Line 220 scans the shape you will have previously displayed and paddle-defined in terms of its block-size.

Line 225 has the shape-sequence creation quit if all the desired shapes are done.

Line 240 shifts the presently displayed shape one bit right for each dot of movement required according to the step-size. A step-size of 2 would cause 2 logical shifts here.

Line 245 increments ST, the variable that will be POKEd into $7, the shape # holder, if another shape is required.

Line 250 keeps the cycle going until an earlier line pulls the program out of the loop and sends it to line 300.

Line 300-310 saves the block-shape sequence table.

It should be noted that **a block-shape sequence creation requires that the actual shape inside the block-shape is at least (7 times step-size) dots from the right edge of the defined block of the block-shape,** if a block-shape sequence creation is to be successfully accomplished. The reason is simple: there must be shifting room! You can't do shift operations on bytes you're not even including in your block's definition.

Now let's return to the front end of this chapter and again look over the printed-out view of the 2 shape tables MANB and MANC (little walking people are the shapes). Notice that this type of sequence creation involves many **different** shapes to begin with, while automatic sequence creation involves making many shapes out of one shape and all the shapes end up identical (except for horizontal positioning). Now imagine the top shape in each column being identical to each shape under it --- only one shape type per sequence. This is what automatic sequences are all about. A flying saucer simply moves in some direction (let's say rightwards, for now) without changing its shape. Much animation today is of this type, which is why I know you'll appreciate A of 28A and 1 of 28B.

Notice again the relative jump taken by MANB and MANC shapes. If MANB shapes were in color, they'd be in trouble, since a 1-bit shift throws off colors. MANC shapes, on the other hand, when used in block-shape sequence routines such as the one in D of 28A, would have no color problems, since 2-bit displacements are in harmony with the "even or odd" Apple color scheme.

Anyway, all you need to do to run 1 of 28B (and A of 28A is run about the same) is to load a block-shape into the program, define it with the paddles, and answer the inputs about shape #s, etc. Then it's just Presto Bismo and you've got yourself a block-shape sequence.

Okay, now back up a couple of pages and review what lines 84-9Ø do in TEST H. There was no way to tell you about lines 91-129 without first letting you have a bit of context about shifting and block-sequences and hi-res graphics bytes. You should be about ready to understand these lines at this point --- we'll see:

Line 91-93 "zeros" the bit 6 and bit 7 flags. I've named $8 the bit 7 flag and $CF the bit 6 flag. All I mean by flag is that I intend to save information about the 6th and 7th bit of each block-shape byte I encounter in this shift-subroutine. I can't save information in the bytes themselves --- we must preserve their patterns and integrity, and possibly even their virginity, if our end result is going to be to shift the shape over 1 bit --- intact and unscarred. Notice that this shifting subroutine only shifts shapes over 1 bit, never more. If you want a 2-bit shift, you need to call the routine twice. (From BASIC, in programs like 1 of 28B, this would be CALL 2186. In mach. lang., this would be JSR $88A.)

Line 94 loads the byte (from the block-shape) to be shifted into the accumulator. The way it's indexed with Y is that originally Y gets loaded with HL (in line 9Ø), but at the end of the routine (line 113) we determine if it's reached HR yet (Y gets incremented at 115 each cycle).

Line 95 shifts the current byte to the left in its structure, but to the right on the screen.

Technically speaking, this is called a ROTATION. ROL means that the 7 bit goes into the carry bit and the carry bit goes into the Ø bit. In effect this makes it a temporary 9-bit byte rotating within itself. In regular shift operations, a Ø is input into the vacant bit (caused by shifting), rather than the carry bit. We call this type of animation **shift** animation because **rotation** animation would be a **very** misleading term!

Line 96 puts this shifted (rotated) byte right back into the address it found it at.

Line 97 branches to line 99 if bit 7 had been 1 before it was shifted. In line 99 the "bit 7" flag ($8) is set to signify that bit 7 was a 1 before it was shifted. (Bit 7 shifts into the carry flag (C) during ROL, therefore BCS (branch on carry set) would be the way to know about the way the byte's bit 7 used to be before shifting.)

Line 98 if bit 7 used to be a 0 then the carry flag would also be 0, and detectable by BCC (branch if carry clear). On clear carry the program would go to line 100.

Line 99 sets "bit 7" flag signifying that bit 7 used to be 1.

Line 100 compares the shifted byte in the accumulator with 128 ($80). Any number 128 or larger means that bit 7 is **now** (after shifting) a 1, which means that bit 6 was a 1 before the shift.

Line 101 if bit 6 used to be 1 then we're sent to line 103 where we set the "bit 6" flag.

Line 102 if bit 6 used to be 0then we're made to skip line 103 and go to line 104 where the routine continues.

Line 103 sets "bit 6" flag ($CF) to show that before shifting bit 6 was a 1.

Line 104-105 loads accumulator with the bit 7 flag and tests it --- if it's NOT set (not equal to zero is the criterion of the BNE instruction) then the BNE is ignored and the current byte is stuck into the accumulator and the #7 bit is changed back into the 0 it used to be before shifting (this will happen in lines 106-108).

Lines 106-108 load current byte, logically AND this with 127 ($7F), and store it back in its place. The way AND works is that when it compares two bytes, it leaves a 1 in only those bits that were 1 for **both** numbers. The result gets stored in the accumulator. The result of ANDing with 127 is that all bits get ignored except bit 7, which gets set to 0. Look at the binary version of 127 and you'll see why. It's 01111111. All one's that get compared to bits 0 through 6 will stay ones. All zeroes will stay zeroes. (Remember to read the most significant bit as being the one on the left.) Bit 7 of any byte, whether 1 or 0, will turn to 0when ANDed with bit 7 of 127 which is 0. Again, the reason we want the high bit zeroed here is because it **was** 0 before the shifting and we want to restore its previous condition in order to keep colors intact (bit 7 is color bit).

Line 109 sends you to line 113 since bit 7 is now fully taken care of.

Line 110-112 where we were sent if lines 104-105 found the bit 7 flag set (meaning it was 1 before the shifting). All we do in these lines is put the 1 back in bit 7 the way it used to be. We simply load in the current hi-res byte, ORA it with 128 ($80), and store it back where it came from, with a gentle pat on the assembler. The way ORA works is to fix it so that if either one of the 2 operand's bits are 1 then the result is equal to 1. The binary of $80 or 128 is 10000000 --- the **opposite** of the 127 we used in the last logical operation. The net effect is again ignoring bits 0 through 6 but setting bit 7 to 1 no matter what. ORA is often used to set things to 1 and AND if often used to set things to 0 and it's probably obvious why. EOR is the third logical 6502 instruction, and does complementing or comparisons --- such as we've already illustrated in the EOR used to XDRAW in our drawing routines.

Line 113-114 checks Y to see if we've shifted our way from HL to HR yet. If we have, we're sent to line 120 to move up a line.

Line 115 increments Y so that we'll next be shifting the next byte to the right.

Line 116 clears carry flag so line 118 will do its comparison with a clean slate.

Line 117-118 compares the "bit 6" flag ($CF) with 1, which means it sees if the bit 6 flag is set. If it is set, it means $CF was a 1 but also that before shifting, bit 6 of the current byte was a one. The CMP instruction sets carry (C) if the accumulator (with the bit 6 flag in it) or = the data it's compared to (immediate number #$1).

Line 119 starts us back at line 91 to continue on with this line, shifting more bytes.

Line 12Ø-121 moves us up a line.

Line 122-123 finds out if we're trying to go higher than line Ø (**lower** lines are Ø --- lower means the downwards direction), which is a no no; if such a no no is being attempted you're sent to line 126 where you return from the shifting subroutine and go without supper and say 3 Hail Marys.

Line 124 looks to see if you've gotton to the defined top of your block-shape (VT). If you have, you return from your subroutine.

Line 125 if you haven't reached VT you're sent back to begin the shifting of bytes in the next higher line (higher in position, lower in vert. coord.).

Line 126 returns you to the routine you came from. It should be noted that each time you're sent back to line 91 to continue shifting rightwards through the line you're on, you'll be arriving there with the carry bit set or unset depending upon lines 117-118. These lines set the carry if the bit 6 flag had been set before the shift. As it happens, when we shift out of one byte and sideways into the next, we want bit 6 from the 1st byte to determine (1 or Ø) the Ø bit of the 2nd byte. This would allow a shape to smoothly pass through all the horizontal bytes in the computer without distortion, as long as this bit 6 procedure was combined with the other features of this shift program, such as the bit 7 procedure and the ANDing, ORAing, and flagging at specific points.

The reason a set carry bit will stick a 1 into bit Ø of the next byte has to do with how ROL works. Consult your assembly book again. See how the carry (C) goes into bit Ø? Well, that's the trick here. No one ever said that the carry couldn't get set by comparing a bit 6 flag with 1 before re-entering the ROL section again!

Later in this book we'll see how ROL or ROR can be the basis of **logical shift animation**. It's slower than sequences of block shapes and erase/draw routines, especially if YTABLE is used. But it's really quite nice in 2 basic ways'

1) you need only 1 block-shape to move 1 or 2-bit moves during animation

2) it's simpler to use and takes less memory

A section on the use of YTABLE is included in Chapter 31, and a program on this speed aid is found on disk C --- it's #4. There'll also be more animation chapters in this book, relating to not only sequences but to shifting, Applesoft, assembly, etc.

9C. HPLOT SEQUENCES

Hplot-shape sequences are less difficult to program than block-shape sequences. Hplot-shapes take up less room and are easier and faster to draw than any other type of shape. And if an hplot-shape is large and composed of only a few long straight lines, then hplot-shape sequence animation may be faster than block-shape animation. For small shapes block-shape animation would beat hplot-shape animation, even if only a few straight lines are involved. Hplotting commands take longer to execute than drawing and erasing block-shapes, but it depends on the # of bytes you need to handle in a block-shape. If you're considering which method to go for, block-shapes are very rapid when 60-100 bytes are involved, but get proportionately slower when 500-1000 bytes are involved.

This system deals with hplot-shapes in the following programs on disk 28B: 6,7,8,9,A.

In 7 of 28B there is an hplot animation sequence. Try running it. It's Applesoft run, and therefore slow. Use the hplot-shape sequence table TRIANGLE and the inputs the program tells you to give.

It should be noted that if a shape is needed that will move without changing shape, what you need is only one hplot shape --- no sequences are necessary. TRIANGLE is a shape table that is truly sequential because the shapes are all different. Actually, each one is simply bigger than the one before. There are 10 shapes.

An hplot shape is merely a # (representing the # of points there are coordinates for), followed by the 3-byte point coordinates. If desirable, one could easily get the 3 bytes per point down to 2 bytes per point, just by eliminating the possibility of traveling past or drawing past hor. coord. 255. If you keep the hor. coord. under 256 then no high byte is necessary, and if you plan to make sure you'll never need any hor. coord. greater than 255 then you could eliminate all consideration of the hor. high byte totally and just keep it permanently at 0.

Our routines will go both ways with this idea --- nothing will be drawn beyond 255 in the routines, but the hplot-shape tables will include low and high hor. bytes as well as vert. byte. Our hplot-shape drawing routines are all ready to use hor. high byte, but in our animation programs (for hplot-shapes) we don't check hor. high byte. TEST Q is an hplot-shape animation source-code. To add high-byte checking we'd have to add:

```
58  LDA  (BASL),Y
59  ADC   $CF
60  ADC   $CF
61  STA  (BASL),Y
62  INY
63  LDA  (BASL),Y
64  BEQ  GO-ON
65  DEY
66  LDA  (BASL),Y
67  CMP  #$F
68  BCS  RTR
69  INY
```

```
7ØGO-ON  INY
71  DEC  $6
72  BNE  START
73  RTS
74RTR  PLA
75  PLA
76  RTS
```

(continue with TEST Q by moving every line from the present 72 line on up,up 5 line #s; the present 72 would be 77, the 73 would turn into 78, etc., all the way up until line 133 turned into line 138).

Don't mess around with this hor. high byte checking unless you're sure you need 279 hor. units to play with rather than 255.

At present 8 of 23B draws only up to 255 horizontally. To get it up to 279 you need to retype the program on your own disk, after listing it out with E of 23B and jotting it down, and this time multiply the lines that read the PDL #Ø (for hor. coords.) by 1.Ø9.

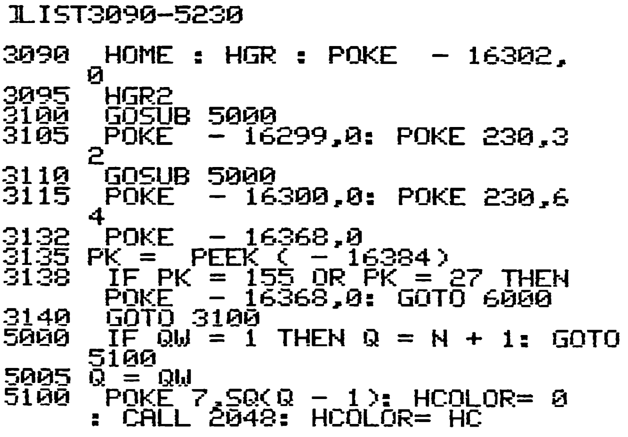
The 9% advantage of the extra 24 dots sideways may seem valuable to some, but they also slow down the program (notice the extra lines TEST Q, above, would need to deal with high byte checking).

Another limit in 8 of 28B is that if your PDL#1 goes above 159 (thereby going from mixed to full-screen graphics, coordinate-wise), then it will continue reading that PDL forever until you turn the PDL more counter-clockwise to bring down the coordinate. Mixed screen was used here so that you could monitor your coordinate #'s.

It should be clear now that one doesn't need more than one hplot-shape to do animation unless the shapes are to be dissimilar.

Let's look now at the basic Applesoft animation program for hplot-shapes, one of the subroutines to be found in 7 of 28B. This animation subroutine uses TEST O (CALL 2Ø48) as its simple hplot-shape-drawing mach. lang. routine. This routine is CALLed to erase in line 51ØØ. But let's look at the program:

PRINTOUT #22



```
]LIST3Ø9Ø-523Ø

3Ø9Ø  HOME : HGR : POKE  - 163Ø2,
     Ø
3Ø95  HGR2
31ØØ  GOSUB 5ØØØ
31Ø5  POKE  - 16299,Ø: POKE 23Ø,3
     2
311Ø  GOSUB 5ØØØ
3115  POKE  - 163ØØ,Ø: POKE 23Ø,6
     4
3132  POKE  - 16368,Ø
3135 PK =  PEEK ( - 16384)
3138  IF PK = 155 OR PK = 27 THEN
      POKE  - 16368,Ø: GOTO 6ØØØ
314Ø  GOTO 31ØØ
5ØØØ  IF QW = 1 THEN Q = N + 1: GOTO
     51ØØ
5ØØ5 Q = QW
51ØØ  POKE 7,SQ(Q - 1): HCOLOR= Ø
     : CALL 2Ø48: HCOLOR= HC
```

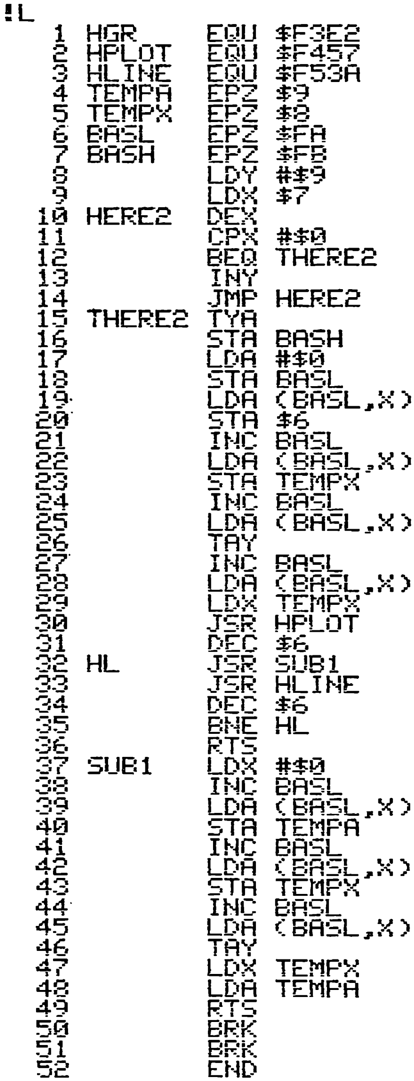
```
51Ø1  ON QL GOSUB 63Ø1Ø
512Ø QW = QW + 1: IF QW > N THEN
     QW = 1
513Ø  POKE 7,SQ(QW): HCOLOR= HC: CALL
     2Ø48
52ØØ  FOR V = 1 TO Z(QW): NEXT
523Ø  RETURN
```

Line 3Ø9Ø-3Ø95 initializing hi-res page 1 and 2.
Line 31ØØ gosub 5ØØØ, the hplotting routine (we're viewing and drawing on page 2 because of HGR2).
Line 31Ø5 view page 2, draw on page 1.
Line 311Ø gosub hplotting routine.
Line 3115 view page 1, draw on page 2.
Line 3132 reset keyboard stroke so line 3135 will work.
Line 3135 read keyboard.
Line 3138 if ESC was hit, exit.
Line 314Ø cycle back to beginning of animation.
Line 5ØØØ N is # of shapes in sequences, QW is the # of the shape now being drawn, Q is one more than the next proper shape # to erase; if we're going to be erasing a shape and then drawing a shape whose # is 2 greater (which is how this routine operates), then when we're on #1, it means that the proper shape # of the shape to erase is not #Ø, but #N, which is the highest numbered shape in the sequence.
Line 5ØØ5 dump the new shape # of the shape you're on into the "shape (+1) to erase" variable Q.
Line 51ØØ put the # of the shape to erase into $7, the shape # holder. If QW was 1 in line 5ØØØ and Q=N + 1 (let's say N was 1Ø), has made Q be 11, then SQ(Q - 1) will put sequence shape #1Ø into $7. The "chart" used to help explain block-shape sequences (2-page) should be reviewed if the juggling of shape #'s is confusing here. What's meant by SQ(Q - 1) is this: an array has been formed that tells us the order of the shape #'s you want in your sequence. SQ with 1 dimension was the array used, and proper DIMensioning has been done. The shape # order may be consecutive or mixed. Note that no XDRAWing is possible so HCOLOR is set to black before the routine is CALLed, and then restored afterwards to the chosen color (default = white).
Line 51Ø1 if flag QL = 1 then we go to 63Ø1Ø where motion stops until keyboard is hit. This flag was set earlier when the program asked you if you wanted "stop-action" animation.
Line 512Ø the shape # we're on is incremented and if we go beyond the last shape # we start over.
Line 513Ø erasing is done so drawing must begin: put shape seq. # in $7 as before, only this time let it be the # we're **on**; draw that shape, by hplotting.
Line 52ØØ runs the delay loop, which may be individualized for each shape in the sequence by use of the special array previously used in the inputting of delay-loop times for each shape in the sequence.
Line 523Ø returns us from this subroutine.

The above Applesoft program gives slow animation which is not intended for actual animation routines such as 9 or A of 28B. Its intention is to allow you to test out an hplot-shape sequence of your own easily, with or without the valuable feature: stop-action mode.

Now let's look at the source-code program TEST O, the LISA-EXECable text file that created, once assembled, the binary file TEST O (CALL 2Ø48):

PRINTOUT #23



```
!L
  1 HGR     EQU $F3E2
  2 HPLOT   EQU $F457
  3 HLINE   EQU $F53A
  4 TEMPA   EPZ $9
  5 TEMPX   EPZ $8
  6 BASL    EPZ $FA
  7 BASH    EPZ $FB
  8         LDY #$9
  9         LDX $7
 1Ø HERE2   DEX
 11         CPX #$Ø
 12         BEQ THERE2
 13         INY
 14         JMP HERE2
 15 THERE2  TYA
 16         STA BASH
 17         LDA #$Ø
 18         STA BASL
 19         LDA (BASL,X)
 2Ø         STA $6
 21         INC BASL
 22         LDA (BASL,X)
 23         STA TEMPX
 24         INC BASL
 25         LDA (BASL,X)
 26         TAY
 27         INC BASL
 28         LDA (BASL,X)
 29         LDX TEMPX
 3Ø         JSR HPLOT
 31         DEC $6
 32 HL      JSR SUB1
 33         JSR HLINE
 34         DEC $6
 35         BNE HL
 36         RTS
 37 SUB1    LDX #$Ø
 38         INC BASL
 39         LDA (BASL,X)
 4Ø         STA TEMPA
 41         INC BASL
 42         LDA (BASL,X)
 43         STA TEMPX
 44         INC BASL
 45         LDA (BASL,X)
 46         TAY
 47         LDX TEMPX
 48         LDA TEMPA
 49         RTS
 5Ø         BRK
 51         BRK
 52         END
```

In this assembly lang. program, the Apple routines to do an HPLOT ($F457) and an HLINE ($F53A) routine are used. In HPLOT you enter with X register = hor. lo. (screen coord.) and Y register = hor. hi. and the accumulator = vert. The routine calls HPOSN, which sets $E0 = X, $E1 = Y, $EZ = accumulator; then it sets up the internal cursor, using $E6 (230 dec.) to tell it what page to operate on, and then it sets $1C to the contents of $E4 (where the color code, gotten from the color code masking table at $F6F6, is stored). The **internal cursor** is the combination of addresses that follow: $1C, where the color masking byte (shifted for odd addresses) is stored; $26,$27, where the left most vertical byte's address is stored (always hor. byte #0) with lo. 1st, hi. 2nd; $E5, the integer part of the hor. screen coord. divided by 7; $30, the bit position from the bit position table at $F5B2, which corresponds to the remainder of the horizontal coordinate divided by 7.

The **external cursor** is the $E0,$E1,$E2,$E4,$E6 all mentioned above. These terms (internal and external cursor) were coined by one C. K. Mesztenyi to help make hi-res graphics more understandable. Judge them for yourself.

After HPLOT calls HPOSN it goes to PLOT, which uses the internal cursor data and does the following:

```
LDA  $1C    (color)
EOR  ($26),Y
AND  $30
EOR  ($26),Y
STA  ($26),Y
RTS
```

In this set of instructions, $26 is responsible for the plotting happening at the correct vert. coord., and when I say $26 this **includes** $27, since the lo. byte is in $26 and the hi. byte is in $27. This is called post-indexed indirect addressing, and is of the form **(addr),Y**, which can be even better illustrated by (addr. +1, addr.),Y. The first of these 2 ways of addressing is really happening, so an instruction like EOR ($26),Y really means EOR ($27,$26),Y, and when commas are used between addresses in the form $27,$26 it means that the hi. byte is 1st and the lo. byte is 2nd and the actual address is found by multiplying the 1st address byte by 256 and adding it to the 2nd address byte.

The Y in the post-indexed indirect addressing above is gotten from $E5 which is the hor. screen coord. divided by 7, which changes the screen coord. into the proper hi-res hor. byte (there are 40 hor. bytes in each line on the hi-res screen --- from 0 to 39; there are 280 hor. coord. units so it's obvious that dividing by 7, since there are 7 (visible) bits per byte, is going to give you 40.).

ANDing with $30 makes plenty of sense once you check out the bit position table:

```
$F5B2:  $81 which in binary form is: 10000001
$F5B3:  $82 which in binary form is: 10000010
$F5B4:  $84 which in binary form is: 10000100
$F5B5:  $88 which in binary form is: 10001000
$F5B6:  $90 which in binary form is: 10010000
$F5B7:  $A0 which in binary form is: 10100000
$F5B8:  $C0 which in binary form is: 11000000
```

If you recall, ANDing has the following truth table:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

This means that when the accumulator (which has been loaded with the color byte ($1C) and EORed with the hi-res byte being PLOTted into) gets ANDed with the data in $30, the color byte determines the color and the bit table # determines that one and only one bit gets turned on, since ANDing with 0 always gives a 0 result, and all visible bits in the bit table # are 0 except one. So AND $30 turns on the correct bit within the hi-res screen byte you're dealing with.

But why EOR ($26),Y, you ask? Because this is how one applies the color masking table byte in $1C to the hi-res byte in question, to make the dot be the right color (which has to do with its position on the screen). Look at the color masking table now:

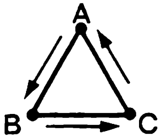
$F6F6: $00 = black1 = binary 00000000
$F6F7: $2A = green = binary 00101010
$F6F8: $55 = violet = binary 01010101
$F6F9: $7F = white1 = binary 01111111
$F6FA: $80 = black2 = binary 10000000
$F6FB: $AA = orange = binary 10101010
$F6FC: $D5 = blue = binary 11010101
$F6FD: $FF: = white2 = binary 11111111

You can probably see why certain colors are good for even coordinates (hor.) and others are good for odd coordinates, when vertical lines or points are plotted. Notice how every other bit is on in green, violet, orange, or blue colors. But enough technicalities --- we don't really need these very much anyway --- what's needed is a good sense of how to **use** the available routines in Applesoft, and the routines you and I create to make hi-res graphics and animation faster, easier, and more convenient.

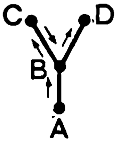
The other routine, HLINE, is entered with the hor. lo. in the accumulator, the hor. hi. in X, and the vert. coord. in Y. This defines the point you hplot **to.** The point you're hplotting **from** is the **last** point hplotted **to** OR the last point plotted on the screen. Let's see how this works now in TEST O, which merely draws on hplot-shape. Consult the list-out previously given.

From line 8 to line 18 we're back to the same old method of table using where shape #1 starts at $900, #2 starts at $A00, up to #23 starts at $1F00. Block-shape and hplot-shape tables will basically have the same structure then. Vector tables have their indexes (often unused if you're in mach. lang.) just before their shapes begin, and **as part of the tables.** Block-shapes and hplot-shapes are being indexed within the actual drawing programs themselves. For an even faster drawing routine, POKE in start-of-table address (high byte) at $FB, POKE 0 into $FA, and begin drawing routines with line 19.

Line 19-20 loads the # of points in the shape to $6, where we'll be keeping track of how many points we've drawn to so far --- $6 may be called the **point counter**. Remember that a triangle would be considered to have 4 points, since we're talking about a starting point plus each point drawn **to**. Notice that the starting point in a triangle is used twice: to start from and to end up at. In other figures we need to draw the same line twice to get the shape completed, so points may be used 2 or more times, when needed. Both situations are illustrated below:



4 points, in order: A,B,C,A



5 points, in order: A,B,C,B,D

Also notice that only in an hplot-shape is the first byte in the shape meant to represent the # of points in that shape.

Line 21-23 gets the hor. lo. for use in HPLOT and dumps it into TEMPX. The reason it wasn't dumped into X, where it's needed to do HPLOT, is that X needs to be Ø for the pre-indexed indirect addressing we're doing with LDA (BASL,X) instructions. The "indirect" in this addressing mode refers to the fact that we're not loading anything from BASL ($FA) --- we're loading the accumulator with **the contents of the address which is given in $FA** --- a very indirect route indeed for getting a # into the accumulator. The parentheses refer to "indirectness," or to "the address to be found at this address."

Line 24-26 loads Y with the hor. hi., which is something that can be left out if you intend to use only 255 out of the 279 possible hor. coords. on the hi-res screen. If you did omit this, all programs would have to put Ø into Y before HPLOT routines and put Ø into X before each HLINE routine, and all shape tables would use 2 bytes per point, not 3 bytes as they do now.

Line 27-28 puts the vert. coord. into the accumulator to prepare for HPLOT.

Line 29 pulls hor. lo. from TEMPX and puts it into X where it belongs, now that X is no longer being used for pre-indexed indirect addressing.

Line 30 plots the shape's starting point.

Line 31 keeps track of the fact that one point is now used.

Line 32 jumps us to the "get ready to do HLINE" subroutine from which we return with an RTS.

Line 33 calls the HLINE routine at $F53A.

Line 34-36 keeps track of each point we use and keeps decrementing the point counter down to Ø. If the counter is greater than Ø we loop back to line 32 to HPLOT another line with HLINE. Once the points have been used up and the counter has reached Ø, we leave the drawing routine at line 36.

Line 37-52 the "get ready to do HLINE" subroutine; puts hor. lo. into accumulator, hor. hi. into X, and vert. coord. into Y before returning.

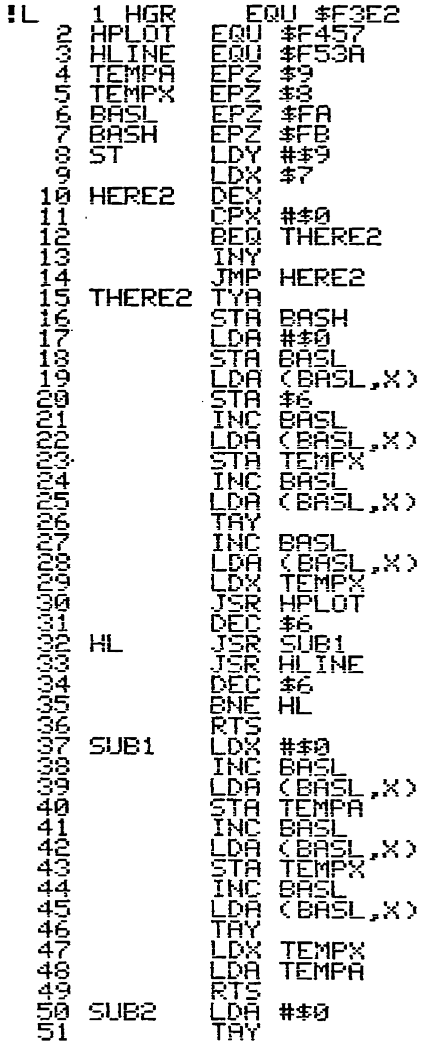
It's important to not lose sight of the fact that one enters HPLOT and HLINE with coordinates in **different** registers:

HPLOT: X = hor. lo., Y = hor. hi., accum. = vert.
HLINE: accum. = hor. lo., X = hor. hi., Y = vert.

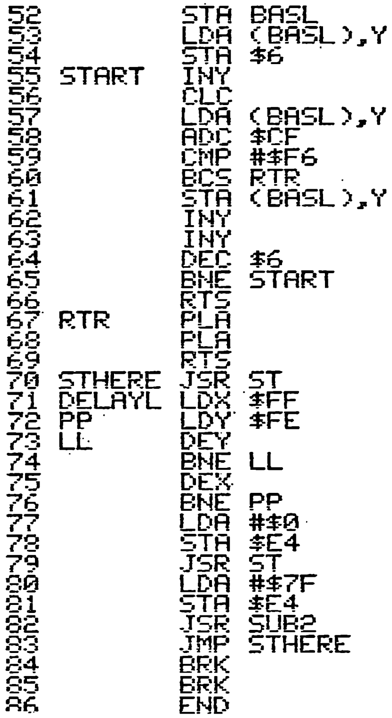
This TEST O (CALL 2048) routine is the basic routine for 6,7, and 8 of 28B.

In 9 of 28B are 2 other hplot-shape routines: TEST P (CALL2160) with a source-code file TEST P, and TEST Q (CALL2186) with a source-code file TEST Q. First let's see TEST P, which is a one-page animation routine for hplot-shapes:

PRINTOUT #24



```
!L
HGR      EQU $F3E2
HPLOT    EQU $F457
HLINE    EQU $F53A
TEMPA    EPZ $9
TEMPX    EPZ $8
BASL     EPZ $FA
BASH     EPZ $FB
ST       LDY #$9
         LDX $7
HERE2    DEX
         CPX #$0
         BEQ THERE2
         INY
         JMP HERE2
THERE2   TYA
         STA BASH
         LDA #$0
         STA BASL
         LDA (BASL,X)
         STA $6
         INC BASL
         LDA (BASL,X)
         STA TEMPX
         INC BASL
         LDA (BASL,X)
         TAY
         INC BASL
         LDA (BASL,X)
         LDX TEMPX
         JSR HPLOT
         DEC $6
HL       JSR SUB1
         JSR HLINE
         DEC $6
         BNE HL
         RTS
SUB1     LDX #$0
         INC BASL
         LDA (BASL,X)
         STA TEMPA
         INC BASL
         LDA (BASL,X)
         STA TEMPX
         INC BASL
         LDA (BASL,X)
         TAY
         LDX TEMPX
         LDA TEMPA
         RTS
SUB2     LDA #$0
         TAY
```



```
52          STA BASL
53          LDA (BASL),Y
54          STA $6
55 START    INY
56          CLC
57          LDA (BASL),Y
58          ADC $CF
59          CMP #$F6
6Ø          BCS RTR
61          STA (BASL),Y
62          INY
63          INY
64          DEC $6
65          BNE START
66          RTS
67 RTR      PLA
68          PLA
69          RTS
7Ø STHERE   JSR ST
71 DELAYL   LDX $FF
72 PP       LDY $FE
73 LL       DEY
74          BNE LL
75          DEX
76          BNE PP
77          LDA #$Ø
78          STA $E4
79          JSR ST
8Ø          LDA #$7F
81          STA $E4
82          JSR SUB2
83          JMP STHERE
84          BRK
85          BRK
86          END
```

Up to line 49 there's only the same hplot-shape-drawing routine we've called TEST O. Let's start examining this file in the order it gets used:

Line 7Ø jumps to line 8 where the shape gets drawn at the coordinates in the table. Line 36 returns us from this subroutine.

Line 71-76 is a standard delay loop, with the "high" byte in $FF and the "low" byte in $FE (these aren't really standard hi./lo. bytes at all).

Line 77-78 puts black (Ø) into the color holding address ($E4).

Line 79 erases the shape.

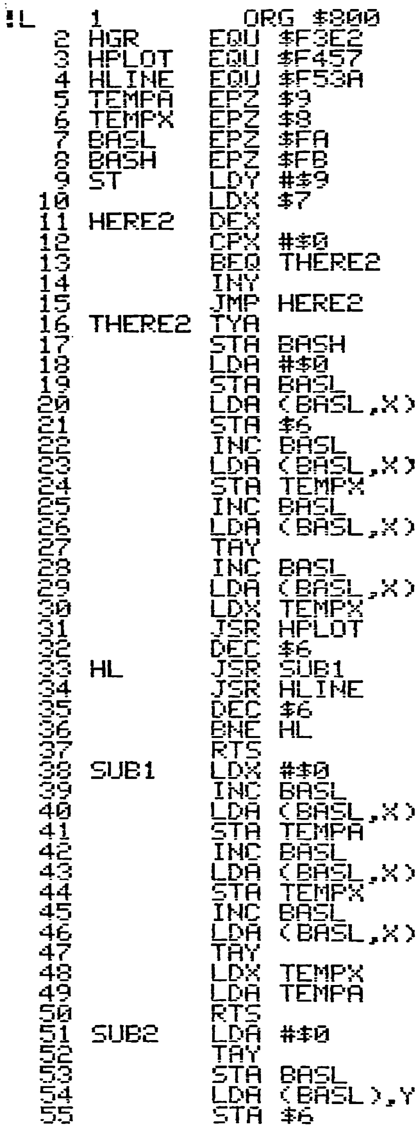
Line 8Ø-81 puts white into the color holding address (#$7F is 127 or binary Ø1111111, which means white).

Line 82 goes to line 5Ø where BASL ($FA) gets restored to Ø; Y gets set to Ø, the # of points gets put into $6, and $CF (the step-size holding address) gets added to the hor. lo. and the vert. shape points coords. If the hor. lo. is greater than or equal to 246 dec., the entire routine is ended and we're RTSed back to whatever called this animation routine to begin with.

Line 83 keeps cycling back to draw and erase the incremented shape until line 59-6Ø determines that the animation has reached the limit of 246.

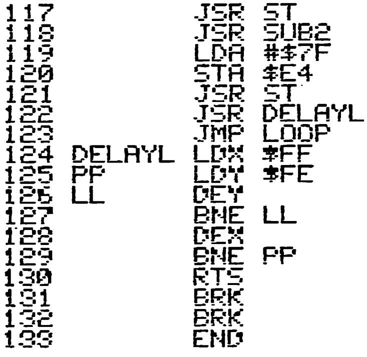
Now let's check out TEST Q, which is a 2-page animation sequence for hplot-shapes, and which assembles a binary file called TEST Q (CALL2186):

PRINTOUT #25



```
!L   1            ORG $800
   2 HGR     EQU $F3E2
   3 HPLOT   EQU $F457
   4 HLINE   EQU $F53A
   5 TEMPA   EPZ $9
   6 TEMPX   EPZ $8
   7 BASL    EPZ $FA
   8 BASH    EPZ $FB
   9 ST      LDY #$9
  10         LDX $7
  11 HERE2   DEX
  12         CPX #$0
  13         BEQ THERE2
  14         INY
  15         JMP HERE2
  16 THERE2  TYA
  17         STA BASH
  18         LDA #$0
  19         STA BASL
  20         LDA (BASL,X)
  21         STA $6
  22         INC BASL
  23         LDA (BASL,X)
  24         STA TEMPX
  25         INC BASL
  26         LDA (BASL,X)
  27         TAY
  28         INC BASL
  29         LDA (BASL,X)
  30         LDX TEMPX
  31         JSR HPLOT
  32         DEC $6
  33 HL      JSR SUB1
  34         JSR HLINE
  35         DEC $6
  36         BNE HL
  37         RTS
  38 SUB1    LDX #$0
  39         INC BASL
  40         LDA (BASL,X)
  41         STA TEMPA
  42         INC BASL
  43         LDA (BASL,X)
  44         STA TEMPX
  45         INC BASL
  46         LDA (BASL,X)
  47         TAY
  48         LDX TEMPX
  49         LDA TEMPA
  50         RTS
  51 SUB2    LDA #$0
  52         TAY
  53         STA BASL
  54         LDA (BASL),Y
  55         STA $6
```

```
 56 START   INY
 57         CLC
 58         LDA (BASL),Y
 59         ADC $CF
 60         ADC $CF
 61         CMP #$F6
 62         BCS RTR
 63         STA (BASL),Y
 64         INY
 65         INY
 66         DEC $6
 67         BNE START
 68         RTS
 69 RTR     PLA
 70         PLA
 71         RTS
 72 SUB3    LDA #$0
 73         TAY
 74         STA BASL
 75         LDA (BASL),Y
 76         STA $6
 77 START3  INY
 78         SEC
 79         LDA (BASL),Y
 80         SBC $CF
 81         STA (BASL),Y
 82         INY
 83         INY
 84         DEC $6
 85         BNE START3
 86         RTS
 87 STT     LDA #$0
 88         STA $C054
 89         LDA #$40
 90         STA $E6
 91         JSR ST
 92         JSR SUB2
 93         JSR SUB3
 94         LDA #$0
 95         STA $C055
 96         LDA #$20
 97         STA $E6
 98         JSR ST
 99 LOOP    LDA #$0
100         STA $E4
101         STA $C054
102         LDA #$40
103         STA $E6
104         JSR SUB3
105         JSR ST
106         JSR SUB2
107         LDA #$7F
108         STA $E4
109         JSR ST
110         JSR DELAYL
111         LDA #$0
112         STA $E4
113         STA $C055
114         LDA #$20
115         STA $E6
116         JSR SUB3
```



```
117                JSR ST
118                JSR SUB2
119                LDA #$7F
120                STA $E4
121                JSR ST
122                JSR DELAYL
123                JMP LOOP
124 DELAYL         LDX $FF
125 PP             LDY $FE
126 LL             DEY
127                BNE LL
128                DEX
129                BNE PP
130                RTS
131                BRK
132                BRK
133                END
```

Lines 1-50 are the same as in TEST P.

Line 87 is where we start when the 2-page flipping routine is called.

Line 87-88 display page 1. (see line 94-95).

Line 89-90 draw on page 2. (see line 96-97).

Line 91 draw the 1st shape.

Line 92 advances shape coordinates 2 **steps**.

Line 93 retreats one **step**, so that the 2nd shape to be drawn will end up only 1 step ahead of the 1st shape, which is on page 2.

Line 94-95 displays page 2. This hex location is called a screen soft switch ($CØ55) and is the equivalent of POKE-16299,Ø, which means display page 2.

Line 96-97 draw on page 1. $E6 (23Ø dec.) is the address for choosing hi-res page 1 or 2. $2Ø (32 dec.) chooses page 1 and $4Ø choose page 2.

Line 98 draws the 2nd shape 1 step from the 1st.

Line 99-100 put black in color address.

Line 101 display page 1.

Line 102-103 draw on page 2.

Line 104 back up one step so that you'll be in the correct position to erase 1st shape.

Line 105 erase shape (1st one drawn, at original coords.).

Line 106 move up 2 steps (study the first 4 columns of the "chart" in Chapter 9B, if this isn't understood).

Line 107-108 put white back into color address. (This could have other colors POKEed into it, if needed.)

Line 109 draw the new page 2 shape (it's the same shape, but at a new position).

Line 110 goes to the delay loop and returns.

Line 111-112 puts black into color address.

Line 113 displays page 2.

Line 114-115 draws on page 1.

Line 116 backs up shape coords. 1 step.

Line 117 erases old shape.

Line 118 moves up 2 steps (on page one still).

Line 119-120 puts white back into color address.

Line 121 draws new page 1 shape.

Line 122 delay loop.

Line 123 loop back to line 99 and continue these erase/draw cycles until lines 61-62 pull you out.

Line 56 get to first point coordinates.

Line 57 be ready for BCS in line 62.

Line 58-60 add 2 steps to hor. lo.

Line 61-62 leave routine if limit (246 dec.) is reached.

Line 63 store addition results in table byte.

Line 64-65 move on to next hor. lo. byte (coord.).

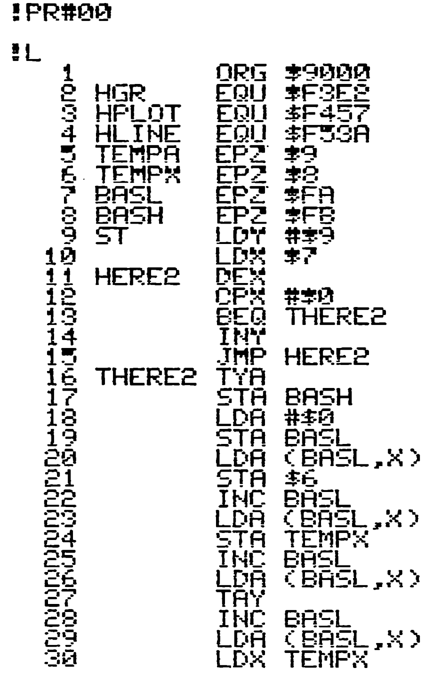
Line 66-68 have we incremented all hor. lo. coords. in the shape? If not, cycle back through; if so then return.

Line 69-70 POP a couple of addresses so you get back to the program that called this animation routine using RTS even though you've JSRed a couple of times.

Line 72-86 this subroutine is much like the previous one except you back up one step rather than advancing 2 in your hor.˙lo. coords. Also, no limits are checked. SEC set's the carry for subtraction in line 80 just as CLC in line 57 cleared the carry for addition in line 59-60.

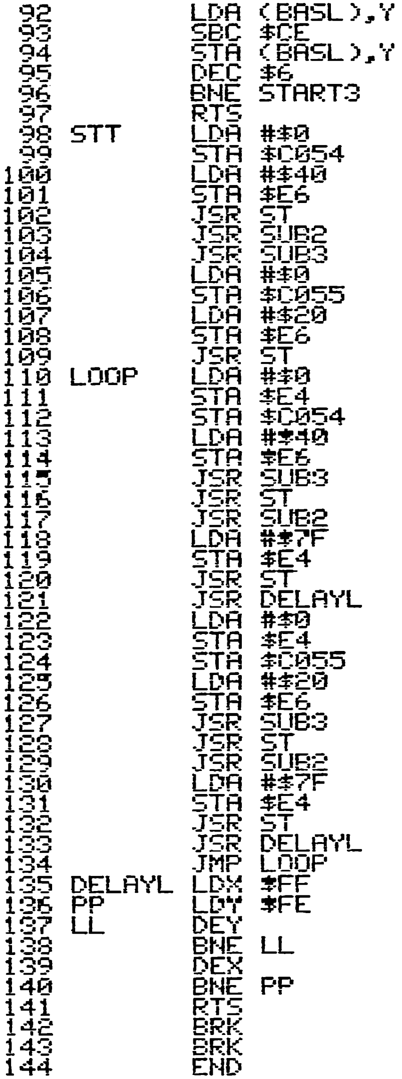
The next source-code file allows diagonal shape (hplot) movement and is called TEST R and it assembles (with a little help) TEST R (CALL37022):

PRINTOUT #26



```
!PR#00

!L
  1         ORG  $9000
  2 HGR     EQU  $F3E2
  3 HPLOT   EQU  $F457
  4 HLINE   EQU  $F53A
  5 TEMPA   EPZ  $9
  6 TEMPX   EPZ  $8
  7 BASL    EPZ  $FA
  8 BASH    EPZ  $FB
  9 ST      LDY  #$9
 10         LDX  $7
 11 HERE2   DEX
 12         CPX  #$0
 13         BEQ  THERE2
 14         INY
 15         JMP  HERE2
 16 THERE2  TYA
 17         STA  BASH
 18         LDA  #$0
 19         STA  BASL
 20         LDA  (BASL,X)
 21         STA  $6
 22         INC  BASL
 23         LDA  (BASL,X)
 24         STA  TEMPX
 25         INC  BASL
 26         LDA  (BASL,X)
 27         TAY
 28         INC  BASL
 29         LDA  (BASL,X)
 30         LDX  TEMPX
```

```
        JSR  HPLOT
        DEC  $6
HL      JSR  SUB1
        JSR  HLINE
        DEC  $6
        BNE  HL
        RTS
SUB1    LDX  #$0
        INC  BASL
        LDA  (BASL,X)
        STA  TEMPA
        INC  BASL
        LDA  (BASL,X)
        STA  TEMPX
        INC  BASL
        LDA  (BASL,X)
        TAY
        LDX  TEMPX
        LDA  TEMPA
        RTS
SUB2    LDA  #$0
        TAY
        STA  BASL
        LDA  (BASL),Y
        STA  $6
START   INY
        CLC
        LDA  (BASL),Y
        ADC  $CF
        ADC  $CF
        CMP  #$F6
        BCS  RTR
        STA  (BASL),Y
        INY
        INY
        CLC
        LDA  (BASL),Y
        ADC  $CE
        ADC  $CE
        CMP  #$B6
        BCS  RTR
        STA  (BASL),Y
        DEC  $6
        BNE  START
        RTS
RTR     PLA
        PLA
        RTS
SUB3    LDA  #$0
        TAY
        STA  BASL
        LDA  (BASL),Y
        STA  $6
START3  INY
        SEC
        LDA  (BASL),Y
        SBC  $CF
        STA  (BASL),Y
        INY
        INY
        SEC
```



```
 92          LDA (BASL),Y
 93          SBC $CE
 94          STA (BASL),Y
 95          DEC $6
 96          BNE START3
 97          RTS
 98 STT      LDA #$0
 99          STA $C054
100          LDA #$40
101          STA $E6
102          JSR ST
103          JSR SUB2
104          JSR SUB3
105          LDA #$0
106          STA $C055
107          LDA #$20
108          STA $E6
109          JSR ST
110 LOOP     LDA #$0
111          STA $E4
112          STA $C054
113          LDA #$40
114          STA $E6
115          JSR SUB3
116          JSR ST
117          JSR SUB2
118          LDA #$7F
119          STA $E4
120          JSR ST
121          JSR DELAYL
122          LDA #$0
123          STA $E4
124          STA $C055
125          LDA #$20
126          STA $E6
127          JSR SUB3
128          JSR ST
129          JSR SUB2
130          LDA #$7F
131          STA $E4
132          JSR ST
133          JSR DELAYL
134          JMP LOOP
135 DELAYL   LDX $FF
136 PP       LDY $FE
137 LL       DEY
138          BNE LL
139          DEX
140          BNE PP
141          RTS
142          BRK
143          BRK
144          END
```

As could be expected, it's much like TEST Q. Why is it at such a high address? Well, since shape tables are always starting at $900, any routine over one page ($100) long is put into high memory so it doesn't mess up any table bytes. Routines start at either $800 (2048 dec.) or $9000 (36864 dec.) and a HIMEM:36864 is in the BASIC program if the higher addresses are used. Otherwise routines start at $800 and end before $900.

You may wonder how each of these programs for hplot-shape animation (8,9,A of 28B) can move shapes in more than 1 direction. Simple, there are POKES in the BASIC driver programs that change ADC to SBC and SBC to ADC and CLC to SEC and SEC to CLC and sometimes DEC to INC or INC to DEC so that shapes may move in different directions! If that isn't clever then my name isn't Don Fudge! (Perhaps that wasn't the best IF-THEN I've ever used.) These BASIC programs are listable via E and F of 28B, so check it out for yourself. If you see dec. #'s in a BASIC program that are being POKEd with 169, it's easier than you think to see how that would affect a binary file (mach. lang.) routine. Use a dec./hex conversion chart to determine that 169 is A9 or immediate addressing of an LDA (load accumulator). Your Apple II Reference Manual and any assembly texts give the instruction set mnemonics and associated hexidecimal opcodes.

Also, to go from hex. to dec., type (let's say you're converting $A197):

```
        CALL-151  (return)
        75: 97 A1 NED2ØG  (return)
        (answer will be here)
```

And to go from dec. to hex., type (let's say you're converting 23456):

```
        New  (return)  (kills program in memory!)
        23456A  (return)
        CALL-151  (return)
        8Ø3.8Ø4  (if ? PEEK (1Ø4) gets you an 8)
        OR4ØØ3.4ØØ4 (if ? PEEK (1Ø4) gets you a 64)
        OR6ØØ3.6ØØ4 (if ? PEEK (1Ø4) gets you a 96)
        (answer will follow)
```

Also, when you want to find what part of a binary file is being changed when there's a POKE 37ØØ2, 169 from BASIC, simply convert 37ØØ2 to hex. as shown above or with a table and then CALL-151 and type the hex # followed by an L (monitor disassembly list-out) command. You'll begin to get the idea of why I decided to POKE what I did into that spot.

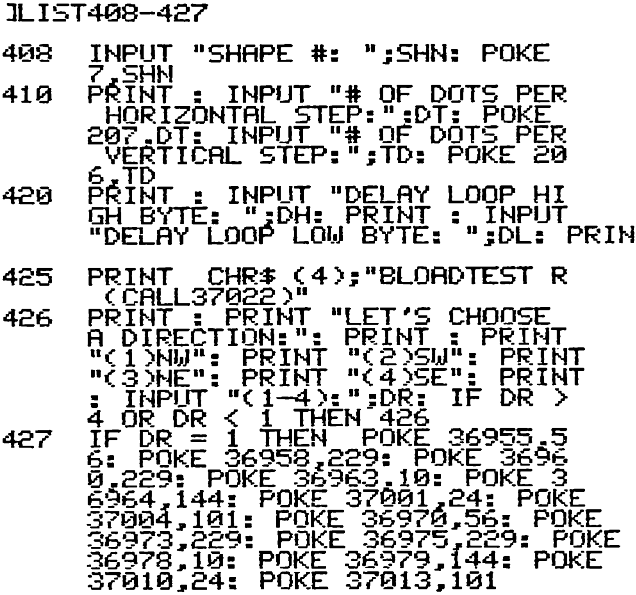
Now look at TEST R. We'll note only **differences** between this diagonal hplot-shape animator and the horizontal hplot-shape animator called TEST Q. Both are 2-page.

In the SUB2 subroutine we add two steps ($CF holds step-size) as usual, but we do it for the vertical coord. as well (line 66-72) and do limit-checking also ($B6 is 182 dec.).

In the SUB3 subroutine we subtract a step as usual, but from both hor. lo. and vert. (vert. coord. handled from line 91-94).

That's all it took! The proper POKEs may be easily found in A of 28B to change this (originally) southeast animation to northeast, southwest, and northwest. Look at a few lines from that program now:

PRINTOUT #27



```
]LIST408-427

408  INPUT "SHAPE #: ";SHN: POKE
     7,SHN
410  PRINT : INPUT "# OF DOTS PER
      HORIZONTAL STEP:";DT: POKE
     207,DT: INPUT "# OF DOTS PER
      VERTICAL STEP:";TD: POKE 20
     6,TD
420  PRINT : INPUT "DELAY LOOP HI
     GH BYTE: ";DH: PRINT : INPUT
     "DELAY LOOP LOW BYTE: ";DL: PRINT

425  PRINT  CHR$ (4);"BLOADTEST R
      (CALL37022)"
426  PRINT : PRINT "LET'S CHOOSE
     A DIRECTION:": PRINT : PRINT
     "(1)NW": PRINT "(2)SW": PRINT
     "(3)NE": PRINT "(4)SE": PRINT
     : INPUT "(1-4):";DR: IF DR >
     4 OR DR < 1 THEN 426
427  IF DR = 1 THEN  POKE 36955,5
     6: POKE 36958,229: POKE 3696
     0,229: POKE 36963,10: POKE 3
     6964,144: POKE 37001,24: POKE
     37004,101: POKE 36970,56: POKE
     36973,229: POKE 36975,229: POKE
     36978,10: POKE 36979,144: POKE
     37010,24: POKE 37013,101
```

Line 408 shape # gets POKEd into $7.

Line 410 step-sizes for hor. and vert. get POKEd into $CF (207 dec.) and $CE (206 dec.) respectively.

Line 427 if NW (northwest direction is chosen, all SEC, CLC, ADC, SBS commands and some BCS or BCC commands, and some limit #'s such as #$B6 (186) must be reversed by POKES.

SHAPE EXAMINATION AND EDITING

10

10A. VECTOR SHAPES

There will be no actual vector-shape editing in this system, unless via monitor commands, which is quite simple if you understand vector shapes. The reason this system is downplaying vector shapes should be obvious by now: I've already written a nice vector shape drawing, editing, examining and animating package called Super Shape Draw and Animate with 2 disks and 2 manuals --- very reasonably priced and easy to use. Peelings Magazine noted that it had qualities (Super Shape Draw) that set it apart from all other packages of this type, such as diagonal as well as horizontal and vertical plotting, any amount of move-up commands, and especially **not having the plotting cursor be always one step out of sync. with what you're trying to do.** Our Super Shape Draw let's you see **exactly where you are** at all times, which no other package we know about can do. (How anyone can make decent shapes with out-of-sync. drawing programs is beyond me! **Sure** it was more difficult to program an in-sync. program, but the results of the extra patience and work I feel were worth the effort 1Ø times over.)

Another reason I'm downplaying vector shapes in this package is that block-shapes are faster. Another one is that there are already a lot of vector shape programs available, but few if any decent block-shape and hplot-shape ones, especially with assembly routines.

If you do end up editing a vector shape, then I of 28A will come in real handy to tell you what address contains what byte and what part of the shape each address and byte correspond to. All you need to do is look in your vector tables and then look at your shape and decide which bytes need amending and how. Once you determine what bytes should replace the ones in your shape that are bummers, simply CALL- 151 and change all necessary bytes via *976:3C 3C 24 Ø4 etc., or whatever (and wherever) you need.

Try out I of 28A now. Notice that a byte either affects one or two dots, never more. (It might be an all-move byte and have no visible effects on the screen, also.) The beep you hear will be your shape starting over in its table as it "winks out" the bytes in its shape. The first coord. in a shape is called its reference dot or coordinates, and it may be invisible (non-plotting) or visible.

One thing to notice is that if any dot in any shape you draw ever plots twice, it will have the consequence of leaving a hole in your shape when you XDRAW it, since the screen becomes its opposite (black to white) and then does it again (white to black) at that particular point because of how EOR works.

If you DRAW the shape, on the other hand, it won't matter how many times any given shape-dot is plotted.

108. BLOCK-SHAPE EXAMINATION AND EDITING

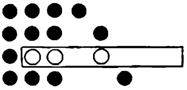
Program 1 of 28A allows you to examine and change a block-shape, or just examine it and exit, if you like. Let's go to that program now and run it:

Boot 28A and choose #1. Choose shape table MANA and don't change disks when asked. Enter shape #1 and use VT= Ø, VB=21, HR=6, HL=4. Read any instructions given. If you hit Q like it says, after editing, you'll be sent to the scan-and-save program where you'll be able to save the repaired shapes. Notice that bytes 5 and 8 are 3, and the rest of the bytes from 1-9 are Ø. A byte of 3 means that the Ø bit and the 1 bit are both on --- and the rest are off. If you'll notice that the byte-cursor is superimposed upon the top 2 head bytes when the 3 appears, and that only the first 2/7 of the byte touches the shape, this should all seem clearer.

Notice that the addresses of those first 9 bytes are 8196,8197,8198,922Ø,9221,9222,1Ø244,1Ø245,1Ø246. Remember, lines are often 1Ø24 (dec.) apart, address-wise. If this isn't clear consult page 21 of the white Apple Reference Manual.

One of the forward arm-bytes that includes 2/3 of the chest is 11 ($ØB in hex. if you were in the monitor). Let's visualize this byte:

11Ø1ØØØØ

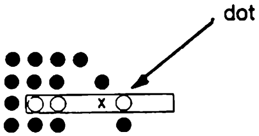


1+2+8=11

Remember, msb (most significant bit) is to the right, which is the reverse of normal byte depictation.

Let's edit the above byte. Speed up the wink-outs with PDL #Ø until you're about 5 bytes from your destination byte. Hit SPACE BAR and turn the knob to its slowest speed. Hit SPACE BAR as you get your pinky poised over the E key. Once the proper byte winks out, hit E quick. You'll be asked for dot color. Choose 3 for white.

Next you'll be moving the paddles to position the drawing dot-cursor correctly. If no dot (blinking) is visible, turn #1 counterclockwise. Move the paddles until your dot is at: (see next page)



(X=old dot-position)
1+2+16=19($13)

Your dot is no longer at bit 3, it's at bit 4. Your data byte is now $13, not $ØB. Hit PDL button #Ø to plot the dot. Answer Y on the next question "Is this dot correct?" and make sure you turn the knob on paddle Ø until the cursor is off the shape before you answer. Then hit PDL#1 and maybe SPACE BAR to continue. When you get the wink-out byte next time around, it will be 16 since there are 2 chest-dots missing. You could have fixed both of those by using the paddles and PDL button #Ø BEFORE you ever hit PDL#1 to **return**. Let's do it now:

Get to the proper wink-out byte and hit E, choose color 3, move the paddles and plot dots (with PDL#Ø button) in **all 3 positions** now. Then hit PDL#1 and wink your way to that byte (8325 dec. address) and notice the 19 (dec.) in it. See? This stuff isn't so hard.

Hit Q when the cursor is in its lowest line and input a 1 to answer shape #, and hit RETURN. You'll be sent to the scan-and-save program. Don't bother to save the shape, but **do** bother to hit SPACE BAR during entry --- your shape, in its edited form, is still there, ready to save.

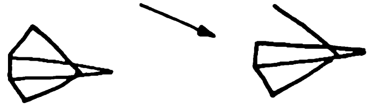
10C. HPLOT-SHAPE EXAMINATION AND EDITING

Program 6 on 28B will examine or edit your hplot-shapes. Run this program and enter T1 for shape table name and 1 for shape #. Then use option 4 to examine the shape.

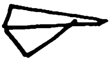
Why do the addresses seem to go only from 23Ø8 to 2331? We know that the first byte of tables (for shape 1) is at 23Ø4, but it's the point #. But 23Ø5-23Ø7 should contain the first point. Why aren't those addresses shown? Because if you'll look to the right, you'll see that from 23Ø8-231Ø is the address for line 1 and point 2. Wink-outs are for each line and not for each point. The points given are the **endpoints** for each line # given. Point 1 is **not** an endpoint, so is not used.

Hit SPACE --- then hit it again. The wink-outs will stop temporarily. When the shape looks like this hit E:

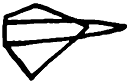
original:



Here's the result you'll get:



As the instructions say, when the line preceding the point to move winks out, hit E and both the line before the point and the line after it will disappear. Now hit any key to continue, use the paddles to find a better point for those 2 lines to meet at; and then hit PDL#Ø. Perhaps it looks like this now:



Answer yes, you got it right, and no, you don't want to do any more editing. Its okay to re-boot once the program tries to have you save the shape.

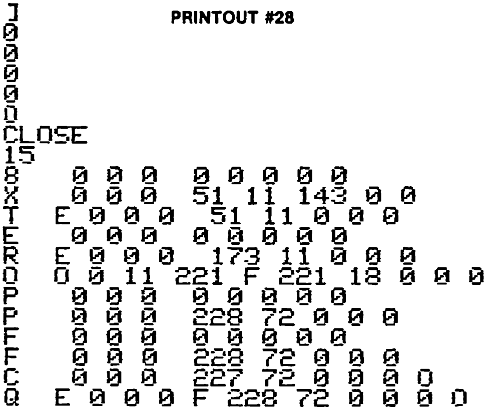
10D. INSTANT GRAPHICS (BLOCK-SHAPES) SHAPE EXAMINATION OR EDITING

This program, 2 on 28A, has already been discussed in Chapter 8D. It draws all kinds of shapes, including user-drawn ones via either PDL - draw features or the J command. The J Command uses keyboard commands to draw by plotting points in the direction of your choice, including diagonals.

There are a few ways of examining and/or editing shapes or scenes drawn with this program.

1) When you save a drawing, the variables being saved for each part of the drawing (13 major variable values get saved for each part of the drawing) will scroll by --- but too fast for you to see. However, if you choose option 8 to See Major Variable Values **before** you choose option 7 (save), then you'll get a chance to see the variables that got saved for each of the parts of the drawing/creation.

PRINTOUT #28



```
]
Ø
Ø
Ø
Ø
O
CLOSE
15
8    Ø Ø Ø  Ø Ø Ø Ø Ø
X    Ø Ø Ø  51 11 143 Ø Ø
T   E Ø Ø Ø  51 11 Ø Ø Ø
E    Ø Ø Ø  Ø Ø Ø Ø Ø
R   E Ø Ø Ø  173 11 Ø Ø Ø
O   O Ø 11 221 F 221 18 Ø Ø Ø
P    Ø Ø Ø  Ø Ø Ø Ø Ø
P    Ø Ø Ø  228 72 Ø Ø Ø
F    Ø Ø Ø  Ø Ø Ø Ø Ø
F    Ø Ø Ø  228 72 Ø Ø Ø
C    Ø Ø Ø  227 72 Ø Ø Ø O
Q   E Ø Ø Ø F 228 72 Ø Ø Ø O
```

Above, the single character lines are the end of the last command saved. "15" is the # of inputs (all of which are not shown). 8, etc. is choosing command option #8 while the program is being saved ("recorded," actually, the **saving** comes later). After that comes X, a slidewall; T E, a small triangle; E, an enclosure; R E, a small rectangle; O O, a tiny circle; P at 0,0 which would be an "illegal" fireworks that would therefore not draw due to its coordinates; P at 228,72, which would draw; F at 0,0; F at 228,72; color change to orange; Q E, a small, filled, orange square. See Chapter 8D to see what each major variable value stands for.

The type of "examination" one does when one looks at the major variable values for one or all shapes in a scene is to look at coordinates, types, sizes, circle or ellipse centers, fill or not. With a set of these values 2 of 28A can draw something and it will be the same each time it gets drawn using these same values. It's easy to save an entire scene in only a few sectors, using this method, and it's possible to draw a scene back, part by part, by the way these variables/values are handled. 2 of 28A will save, retrieve a creation, show major variable values after each shape, create mach. lang. composites of the entire screen, etc.

2) A less involved way of examining/editing shapes is the use of the **monitor** command. By hitting M you will see the coordinates of the floating dot-cursor. We call this **monitoring** the coordinates. If you happen to be "recording" what you're drawing, for a later save via option 7, then M will also give you what step out of a possible 32 you're now on.

To use monitor simply hit M to see what coordinates you're at. To see the coordinates of an already drawn shape, move the cursor onto the major shape points and hit M to monitor coordinates. A square might have corners at (X,Y) 14,50; 14,90; 54,50; 54,90. This square is 40 dots per side.

To use monitor (M) for editing, simply use the M command to find the size of one of the sides of a figure, or to find the diameter of an ellipse or a circle. Then erase the figure by hitting C and B to change color to black (or to whatever background color you're using) and then R F E to erase (or whatever). The example R F E means a rectangle higher than wide, F means fill it, and E means size E. How will you determine what size if you've forgotten?

On the back of the drawing card is a size table so once you know the size of the figure's side you can figure out which size to hit for erasing. Now the given sizes are for **symmetry mode**, during which T, R, W, Q all have the cursor in their centers, as opposed to during regressive mode when the shape's cursors are at their upper left corner.

Now, if you're in regressive mode the size tables don't apply, except for O (circle) or I (ellipse). The way you find out which size is correct for erasing (besides experimentation) is to measure a horizontal side (width) and then notice how far you are from the right edge of the screen. Suppose you are going to erase a W (wide rectangle). Find the hor. coordinate of the 2 lower corners. Let's say that they are 179 and 204. This means the width is 25 and the distance from the right edge (279) to the left side of the figure is 100. So this side is 1/4 of the way from the X,Y of the left side of the figure to the edge of the screen (279). On column 3 of your drawing card you'll see that this would mean that your size must be "S" for "short." This is the size to use when erasing. Normally you can tell that the left hor. coord. and right hor. coord. is approximately whatever it is --- you'd never believe the W (wide rectangle) now under consideration is either as large as 1/2 way to the edge or as small as 1/8 way to the edge, which means that if you know it was drawn in regressive mode, it **must** be size S.

There's no good way to edit out stars (command S) by any other method than drawing black dots on them. Random lines are not really editable --- start over. Slidewalls and such just need to be redrawn in background color --- keep your finger poised over button #0 in order to stop the slidewall at the right point. P (fireworks) gives a different display every time, so use properly-placed black W (wide rectangles) to erase them.

See Chapter 8D for how to erase while in J mode, block-shape drawing via keyboard commands.

10E. SUPER SHAPE DRAW EXAMINATION OR EDITING

In this program, not included in this system, but found on the Super Shape Draw and Animate package, you can throw out any vector shape you want in a shape table and replace it with any other shape you decide to draw, as long as the new shape is not longer than the old.

For in-depth shape examination, there's a program (I of 28A) in this system, and there's an even more dynamic one in our Super Shape Animate disk, which is included in our Super Shape Draw and Animate package. This program not only examines every byte, address, and corresponding hi-res dot on the shape, but also will allow you to edit **parts** of the shape. The general purpose of the program, called "animation editing," is to not only allow you to repair or change shapes, but to construct animation sequences of shapes that will be **dissimilar**.

In the present system that this manual you're reading goes with are 2 programs that create automatic shape (block) sequences (A of 28A, 1 of 28B) and one program (4 of 28A) that can turn vector shape sequences of **dissimilar** shapes into block-shape sequences. Both **automatic** sequence-creators deal with similar shapes only --- you make one shape into 7 or more.

So "animation editing" allows you to knock off parts of a shape and redraw them and resave the shape. This is extremely handy if you're doing walking or running animation. Why completely redraw the shapes?

10F. MONITOR EXAMINATION OR EDITING OF SHAPES

For **vector shape** monitor examination, you first have to know what goes into each of the types of bytes. In Chapter 9 of your Applesoft Manual you'll find how bytes are constructed on page 92, how vector codes relate to hexadecimal bytes on page 93, how hex #s can be changed into binary #s which are then changed back into 2-digit hex bytes on page 94, how shape tables and indexes are constructed on page 95, how table address and HIMEM are handled on page 96 (very poor), saving a shape table (very poor since cassettes are rarely used in shape storing any more --- disks are SO much easier) on page 97, using a shape table and vector shape commands on pages 97-100.

If you want a shape to be different, figure out the bytes to change, CALL-151, examine the bytes to be changed with commands like *930.999 and change the wrong bytes with commands like *947:3C 0B 24.

For **block shape** monitor examination, see Chapter 10B for how bytes and bits can be understood easily, and how binary and hex. are related. (In normal binary bytes, the **most** significant bit is to the left, but in viewing a screen byte, the **least** significant bit is on the left. The decimal values of these bits are 1,2,4,8,16,32,64,128. From this you could see that any hex. byte $7F will have all bits on but the last, invisible one. Add up 1+2+4+8+16+32+64 and you'll see you get 127, which is $7F in hex. The dec. # you'd get if you PEEKed into that $7F byte would be 127. If you POKEd 127 into $900 (address), you'd find $7F if you examined the monitor for what was in $900. Of course, the only way you can POKE 127 into $900 is to convert that address to dec. and get 2304. Then you'd POKE 2304,127. Only in the monitor could you look directly for what was in "$900.")

To change a block-shape byte from the monitor, do the following:

1) Make a drawing of the_dots you want **on**:

●○○●●○●

2) Change the "on" dots to ones:

1001101

3) Add the one bits, according to position value:

1+8+16+64=89

4) Convert to hex:

89=$59

5) Enter the new hex. byte from the monitor and BSAVE the shape table if necessary.

For **hplot-shape** monitor examination, simply look at the first table byte and consider it the # of points. Each group of 3 bytes after that is the hor. lo., hor. hi., and vert. coord. of the point which is hplotted to from the last point:

```
0900-  17  04  00  62  17  00  41
```

Above is the start of a shape with 23 points, which also means 22 lines that get drawn (the first point gets nothing drawn **to** it). The commands above tell the shape to get drawn in a way that would be like a BASIC statement saying to:

```
HPLOT  4,98  TO  23,65
```

This is merely the start of the shape. As you can see, it's quite easy to examine memory and change it for such a simple shape type. Hplot-shapes have very simple constructions.

11

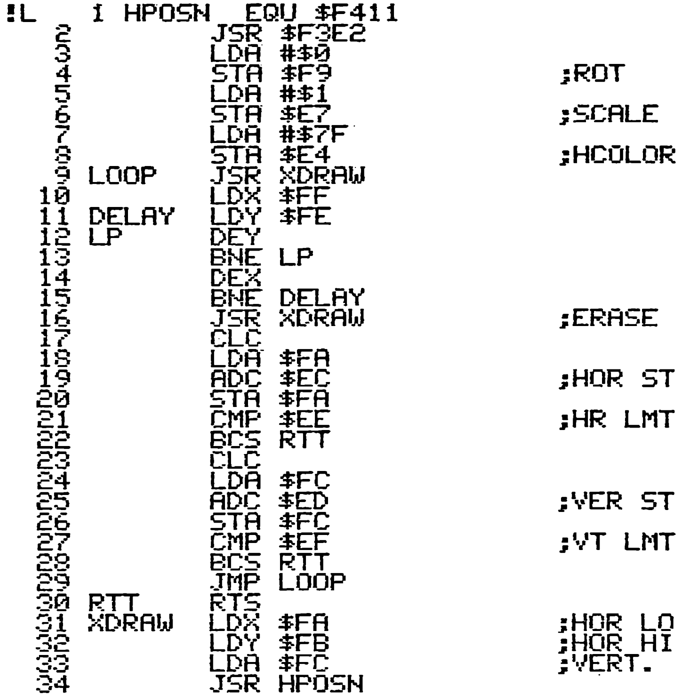
SIMPLE ANIMATION

11A. VECTOR SHAPES

Had I inserted lots of comments into my LISA source-code files, there wouldn't have been room for nearly as many on disk 28C. I'm making up for that by making sure I cover what each source-code line is about. I did put a few comments in TEST C and TEST S, the vector shape animation routines. These we'll get into at this time:

TEST C is the source-code for G of 28A, and the binary file assembled from it is TEST C (CALL2Ø48). It does a 1-page vector shape animation:

PRINTOUT #29



```
!L   1 HPOSN   EQU $F411
     2         JSR $F3E2
     3         LDA #$Ø
     4         STA $F9                   ;ROT
     5         LDA #$1
     6         STA $E7                   ;SCALE
     7         LDA #$7F
     8         STA $E4                   ;HCOLOR
     9 LOOP    JSR XDRAW
    1Ø         LDX $FF
    11 DELAY   LDY $FE
    12 LP      DEY
    13         BNE LP
    14         DEX
    15         BNE DELAY
    16         JSR XDRAW                 ;ERASE
    17         CLC
    18         LDA $FA
    19         ADC $EC                   ;HOR ST
    2Ø         STA $FA
    21         CMP $EE                   ;HR LMT
    22         BCS RTT
    23         CLC
    24         LDA $FC
    25         ADC $ED                   ;VER ST
    26         STA $FC
    27         CMP $EF                   ;VT LMT
    28         BCS RTT
    29         JMP LOOP
    3Ø RTT     RTS
    31 XDRAW   LDX $FA                   ;HOR LO
    32         LDY $FB                   ;HOR HI
    33         LDA $FC                   ;VERT.
    34         JSR HPOSN
```

```
35          LDX $7                     ;SHAPEL
36          LDY $8                     ;SHAPEH
37          LDA $F9                    ;ROT
38          JSR $F65D                  ;XDRAW
39          RTS
40          END
41          BRK
42          BRK
```

A vector shape must have ROT, SCALE, and HCOLOR specified. (You can XDRAW and get along without HCOLOR.)

Line 1 HPOSN is found at $F411.

Line 2 $F3E2 is HGR, page 1 initializing.

Line 3-4 put ROT of Ø into $F9, the ROT holder.

Line 5-6 put SCALE of 1 into $E7, the SCALE holder.

Line 7-8 stick white ($7F is Ø1111111 in binary so the color is white since all visible bits are on) into $E4, the HCOLOR holder.

Line 9 XDRAWS the shape in the XDRAWing routine in line 35.

Line 1Ø-14 usual delay loop.

Line 16 erase (2nd XDRAW erases 1st).

Line 17 get carry cleared so line 23 will work right.

Line 18-2Ø add hor. step size to $FA, hor. low hold.

Line 21-22 compare new hor. lo. with data in $EE, the address holding the horizontal limit; if limit is reached, quit.

Line 23-26 add vert. step to vert.

Line 27-28 see if you've reached vertical limit; if so, quit.

Line 29 loop back to line 13.

Line 31-37 load hor. lo. into X, hor. hi. into Y, and vert. into accumulator; then do HPOSN routine to calculate base address and handle internal and external cursors and color.

Line 38 Applesoft XDRAW routine.

Obviously the BASIC program that calls this routine must have POKEd shape lo. into $7 and shape hi. into $8, Ø into $FB for hor. hi., hor. lo. into $FA, vert. into $FC, hor. limit into $EE, vert. limit into $EF, hor. step into $EC, vert. step into $ED, delay "high" into $FF, delay "low" into $FE.

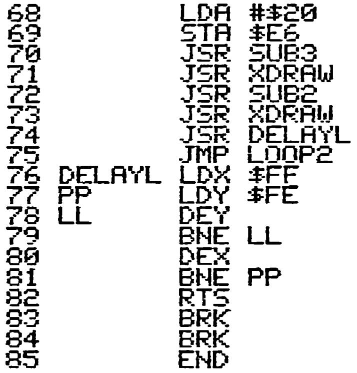
If we were to POKE the hex bytes for opposite opcodes in line 23 and 29 and 26 and 32 and subtract the hor. limit from 255 to get a new one and subtract the vert. limit from 192 to get a new one, we could move in the opposite direction. The opposite of ADC and BCS is SBC and BCC. We'd have to change the opcodes in lines 21 and 27 to SEC, also, so that the SBC instructions would work. G of 28A handles this so you get to go in all 4 directions: up, down, left, right.

TEST S is the source code that creates the binary file TEST S (CALL2125), which is used in H of 28A, which is like G except it's 2-page flipping animation:

PRINTOUT #30

```
]L
   1 HPOSN  EQU $F411
   2 STST   LDA #$0
   3        STA $F9                    ;ROT
   4        LDA #$1
   5        STA $E7                    ;SCALE
```

```
        LDA #$7F                        ;WHITE
        STA $E4                         ;HCOLOR
        RTS
SUB2    CLC
        LDA $FA
        ADC $EC                         ;HOR ST
        ADC $EC
        STA $FA
        CMP $EE                         ;HR LMT
        BCS RTT
        CLC
        LDA $FC
        ADC $ED                         ;VER ST
        ADC $ED
        STA $FC
        CMP $EF                         ;VT LMT
        BCS RTT
        RTS
RTT     PLA
        PLA
        RTS
SUB3    SEC
        LDA $FA
        SBC $EC
        STA $FA
        SEC
        LDA $FC
        SBC $ED
        STA $FC
        RTS
XDRAW   LDX $FA                         ;HOR LO
        LDY $FB                         ;HOR HI
        LDA $FC                         ;VERT.
        JSR HPOSN
        LDX $7                          ;SHAPEL
        LDY $8                          ;SHAPEH
        LDA $F9                         ;ROT
        JSR $F65D                       ;XDRAW
        RTS
STT     JSR STST
        LDA #$0
        STA $C054
        LDA #$40
        STA $E6
        JSR XDRAW
        JSR SUB2
        JSR SUB3
        LDA #$0
        STA $C055
        LDA #$20
        STA $E6
        JSR XDRAW
LOOP2   LDA #$0
        STA $C054
        LDA #$40
        STA $E6
        JSR SUB3
        JSR XDRAW
        JSR SUB2
        JSR XDRAW
        JSR DELAYL
        STA $C055
```



```
68          LDA #$2Ø
69          STA $E6
7Ø          JSR SUB3
71          JSR XDRAW
72          JSR SUB2
73          JSR XDRAW
74          JSR DELAYL
75          JMP LOOP2
76 DELAYL   LDX $FF
77 PP       LDY $FE
78 LL       DEY
79          BNE LL
8Ø          DEX
81          BNE PP
82          RTS
83          BRK
84          BRK
85          END
```

Line 9-26 add not 1 step but 2 to hor. and/or vert. coords. since this is needed for page-flipping routines, as you've seen already --- see earlier chapters.

Line 27-35 subtracts a step from hor. and/or vert. coords. since we need to backup before erasing.

Line 36-44 is the same XDRAWing routine as in lines 31-39 in TEST C.

Line 45 loads ROT, SCALE, and HCOLOR.

Line 46-47 displays page 1.

Line 48-49 draw on page 2.

Line 50 draw shape.

Line 51 move up 2 steps.

Line 52 back up 1 step.

Line 53-54 display page 2.

Line 55-56 draw on page 1.

Line 57 draw shape (both pages now drawn upon).

Line 58-61 display 1 while drawing on 2.

Line 62 back up 1 to erase.

Line 63 erase.

Line 64 move ahead 2 steps to draw.

Line 65 draw.

Line 66 do delay loop from lines 76-82.

Line 67-69 display 2, draw on 1 (Ø needs to be in accumulator for line 67 --- it gets it in XDRAW routine from line 36-44 when line 42 gets Ø from $F9).

Line 70 back up 1 step.

Line 71 erase.

Line 72 move up 2 steps.

Line 73 draw.

Line 74 delay loop.

Line 75 loop back up to line 58 to continue.

11B. BLOCK-SHAPES

Chapter 9B has covered the more complex block-shape animation sequences. Chapter 13 will cover logical shift animation using ROL and ROR.

We'll look now at the simplest type of block-shape animation ---movements up or down or 1 byte or more sideways.

We needn't dwell upon vertical moving. All one need do is erase a shape on the screen, add to or subtract from the VB (lowest vertical coordinate of the block: vertical bottom) and the VT (highest vertical coordinate of the block: vertical top), always changing VT and VB equally. You'd leave HL (hor. left-most coordinate, 0-39) alone and HR (hor. right-most coord., 0-39) as well. With these new coords. you'd again draw and erase the shape, adding delay loops and 2 page flipping as needed.

For a test of left or right hor. movement that is 1 or more bytes per step, let's try out both an Applesoft-run version and an all-mach. lang. one:

Boot 28A and run 3, View Shapes or Animation. Use option 11 to load in block-shape table MANA. Ask for shape #5 at VT=0, VB=15, HR=3, HL=0. Then ask for option #10 and answer no, you don't need the "stop-action" option now. Give a height of 15 and a width of 3 (bytes). Give a starting hor. coord. of 3, a lowest vertical coord. VB of 33 (no special reason), an ending hor. coord. of 36, a step-size of 1, 1 shape in sequence, shape #5, and delay loop of 0.

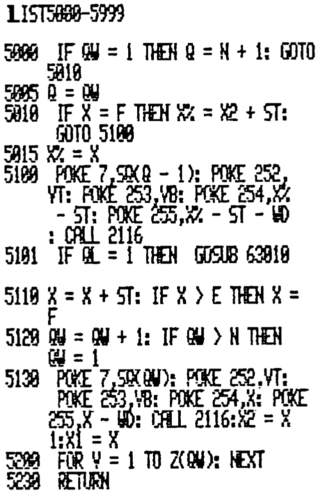
When the routine runs, the animation won't be worth a plugged nickel. Too slow and jerky. You get the feeling for **why** people use mach. lang. rather than Applesoft, and **why** step-size is often only 1 or 2 bits.

Run it again, asking for stop-action, and this time go backwards. Put in 15 for height, 3 for width, 36 for hor. start, 33 for VB, 3 for hor. end, -1 for step-size, 1 for # of shapes, 5 for shape #, 0 for delay loop. Hold down SPACE BAR and REPT but lift REPT at times. Hit ESC once or twice to exit.

Run it again with no stop-action, 21 for height, 2 for width, 2 for hor. start, 34 for VB, 37 for hor. end, 1 for step-size, 9 for # of shapes; 1,6,2,7,3,8,4,9,10, for shape #s, all 0 for delay loops. Notice how woefully inadequate 1-byte (or greater) animation is when it's run by Applesoft. 1 or 2 bit animation looks a lot better. There are times when fast assembly animation can use 1 (or greater) byte moves, but in slower programs it's not too nice. For "walking" animation you need 2-3 bit moves at most. It would be rare to have non-similar shape sequences use greater than 3-bit steps. Perhaps something like grasshopper jumps or frog tongues might be the exception here. Depends upon desired speed and smoothness.

In Chapter 9C we looked at a list-out for an hplot-shape animation sequence program, it was in BASIC --- it was in 7 of 28B. The above program is in 3 of 28A and for block-shape animation sequences. Notice the difference the lines from 3090-3140 were about the same on both, so were shown only for 7 of 28B, and not for 3 of 28A):

PRINTOUT #31



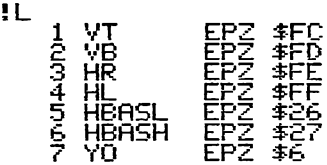
```
LIST5000-5999

5000  IF QW = 1 THEN Q = N + 1: GOTO
      5010
5005 Q = QW
5010  IF X = F THEN X% = X2 + ST:
       GOTO 5100
5015 X% = X
5100  POKE 7,SQ(Q - 1): POKE 252,
      VT: POKE 253,VB: POKE 254,X%
       - ST: POKE 255,X% - ST - WD
      : CALL 2116
5101  IF QL = 1 THEN  GOSUB 63010

5110 X = X + ST: IF X > E THEN X =
     F
5120 QW = QW + 1: IF QW > N THEN
     QW = 1
5130  POKE 7,SQ(QW): POKE 252.VT:
       POKE 253,VB: POKE 254,X: POKE
      255,X - WD: CALL 2116:X2 = X
      1:X1 = X
5200  FOR Y = 1 TO Z(QW): NEXT
5230  RETURN
```

In 7 of 28B there's no limit checking like there is in 3 of 28A in line 5110. In 3 of 28A one needs to both POKE in VB,VT,HL,HR and remember what hor. coords. were being drawn at 2 shapes ago (line 5130). X is now, X1 is last shape, X2 is two shapes ago.

In 3 of 28A in 5010 you need to make sure that if you just got back from the other edge of the screen you'll go all the way back there for erasing --- that's why you kept track of X2 (2 shapes ago hor. coord.). Also, in 5110 you must increment the hor. coord. for the block-shape, whereas in 7 of 28B the shapes are already defined, coordinate-wise, so coordinates may be ignored. An hplot-shape, by its nature, tells us: "here's where I am," but block-shapes and vector shapes both tell us only: "here's the size and shape I am" (a vector shape may be scaled up larger very easily). F is hor. start, E is hor. end, X% is correct erasing coord. HR once ST (step-size) is subtracted. QW is sequence subscript # you're on. Q-1 is correct sequence subscript # for erasing. QL is stop-action flag. Z(QW) is delay loop for that shape.

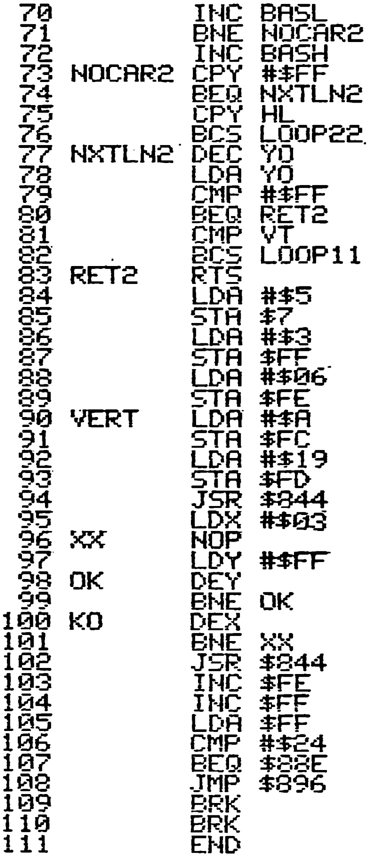
Enough of that --- now let's check out mach. lang. animator that moves 1 byte per step. It's nothing fancy but it may give you ideas. The source-code is TEST A and the binary file is TEST A (CALL2186). The program that drives it is F of 28A. Here's TEST A:

PRINTOUT #32



```
!L
      1 VT      EPZ $FC
      2 VB      EPZ $FD
      3 HR      EPZ $FE
      4 HL      EPZ $FF
      5 HBASL   EPZ $26
      6 HBASH   EPZ $27
      7 YO      EPZ $6
```

```
BASL    EPZ $FA
BASH    EPZ $FB
HPOSN   EQU $F411
        LDY #$9
        LDX $7
HERE    DEX
        CPX #$00
        BEQ THERE
        INY
        JMP HERE
THERE   TYA
        STA BASH
        LDA #$00
        STA BASL
        LDA VB
        STA YO
LOOP1   LDX #$00
        LDY #$00
        JSR HPOSN
        LDY HR
        LDX #$00
LOOP2   LDA (HBASL),Y
        STA (BASL,X)
        DEY
        CLC
        INC BASL
        BNE NOCAR1
        INC BASH
NOCAR1  CPY #$FF
        BEQ NXTLN
        CPY HL
        BCS LOOP2
NXTLN   DEC YO
        LDA YO
        CMP #$FF
        BEQ RETURN
        CMP VT
        BCS LOOP1
RETURN  RTS
        LDY #$9
        LDX $7
HERE2   DEX
        CPX #$00
        BEQ THERE2
        INY
        JMP HERE2
THERE2  TYA
        STA BASH
        LDA #$00
        STA BASL
        LDA VB
        STA YO
LOOP11  LDX #$00
        LDY #$00
        JSR HPOSN
        LDY HR
        LDX #$00
LOOP22  LDA (BASL,X)
        EOR (HBASL),Y
        STA (HBASL),Y
        DEY
        CLC
```



```
70              INC BASL
71              BNE NOCAR2
72              INC BASH
73  NOCAR2  CPY #$FF
74              BEQ NXTLN2
75              CPY HL
76              BCS LOOP22
77  NXTLN2  DEC YO
78              LDA YO
79              CMP #$FF
80              BEQ RET2
81              CMP VT
82              BCS LOOP11
83  RET2        RTS
84              LDA #$5
85              STA $7
86              LDA #$3
87              STA $FF
88              LDA #$06
89              STA $FE
90  VERT        LDA #$A
91              STA $FC
92              LDA #$19
93              STA $FD
94              JSR $844
95              LDX #$03
96  XX          NOP
97              LDY #$FF
98  OK          DEY
99              BNE OK
100 KO          DEX
101             BNE XX
102             JSR $844
103             INC $FE
104             INC $FF
105             LDA $FF
106             CMP #$24
107             BEQ $88E
108             JMP $896
109             BRK
110             BRK
111             END
```

Line 1-83 is the scanning and drawing programs.

Line 84-85 loads shape #5 into shape # address ($7).

Line 86-89 loads 3 and 5 into HL and HR.

Line 90-93 puts 10 into VT and 25 into VB.

Line 94 XDRAWS block-shape.

Line 95-101 delay loop (just enough to keep shape bright, not dull).

Line 102 erases block-shape.

Line 103-104 increments hor. coords.

Line 105-106 checks limits --- have we reached 36? If so start over at line 84.

Line 107 if not at 36, jump to line 94.

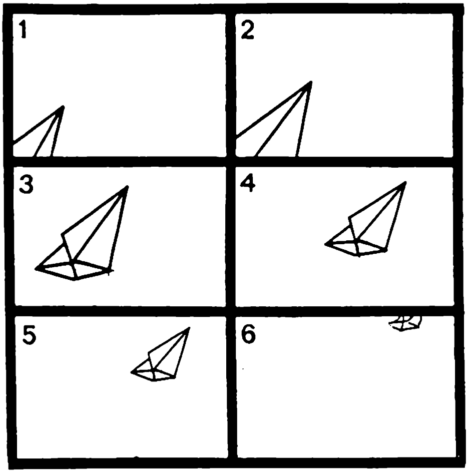
11C. HPLOT-SHAPES

We've already covered simple animation with hplot-shapes in Chapter 9C. It didn't take too much to explore these shapes fairly extensively, since they are constructed so simply. The shape bytes not only say what shape, they say where to put the shape on the screen. No other shapes are like that. You don't XDRAW with these shapes --- erasing is done by drawing with background color.

What if you wanted to rotate or invert an hplot-shape? It wouldn't be too difficult. If you simply took all the Y (vertical) coords. and subtracted them from 192, you'd get the same shape upside down. Or if the shape had no X (horizontal) coord. beyond 255 you could take all the X coords. and subtract them from 255. Or you could do both these things at once. Or you could exchange the X and Y coords., if no coord. was over 191,.and this could be done in addition to other inversions like those mentioned above. Such inversion or interchange routines would be quite simple to tack onto any of the hplot routines in this system. Suppose we were ready to load in a hor. lo. coord. We could JSR to a routine that did this:

```
SEC
LDA  #$FF           ;put 255 into accumulator
SBC   HOR  LO          ;subtract hor. lo. from it
STA  TEMPX ;store new hor. lo. where needed
RTS                              ;return from routine
```

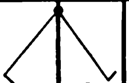
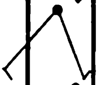
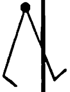
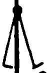
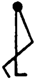
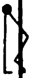
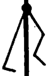
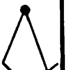
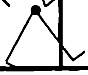
When animating with hplot-shapes, it's okay to simply create a sequence of shapes that are to stay exactly where they are first drawn. This would seem to be the way certain arcade shapes are handled. If you study the animation, you'll see that the shapes often look like they're only being drawn and erased 6 times. Often a good illusion of depth/distance may be created by a simple sequence such as the following 1/2 second (?) hplot-shape sequence, which is very simple to draw, store, and use, and takes very few bytes:



11D. SEQUENCE DIAGRAMS

3-byte wide block shape

MANA

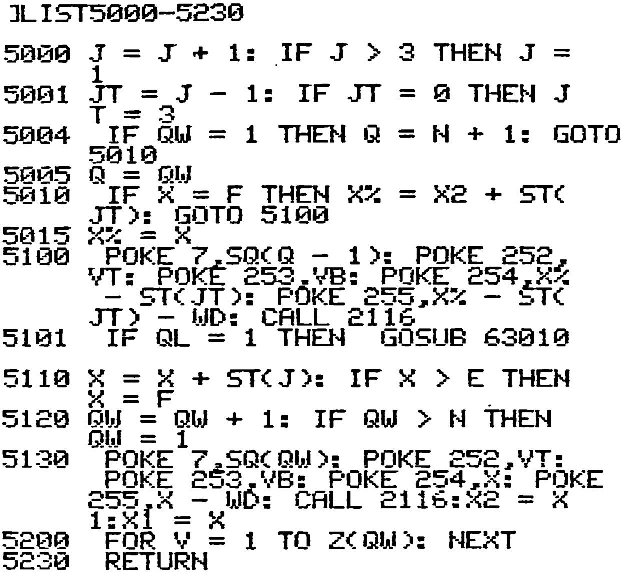
1st byte	2nd byte	3rd byte	sequence #	shape # from MAN	centered on this hor. coord	VT,VB	move HL and HR up by 1?	hor. coords of block-shape boundaries	MANC seq. #
			1	1	35	19,40	yes	28-49	-
			2	6	38	19,40	no	28-49	1
			3	2	40	19,40	no	28-49	2+7
			4	7	42	19,40	yes	35-56	3
			5	3	45	19,40	no	35-56	4
			6	8	47	19,40	no	35-56	5
			7	4	49	19,40	yes	42-63	6
			8	9	52	19,40	no	42-63	-
			9	10	54	19,40	no	42-63	-

Above is a sequence diagram for the block-shape table MANA. The shapes in this sequence were created by loading vector shapes (from the table MAN) into the scanner program (4 of 28A) and saving them as block-shapes. They had to be put at very exact screen coordinates, and PDL-defined one at a time --- there's no automatic sequence creation for dissimilar shapes. 6 of 28A shows a nice slow BASIC-run animation using these 9 shapes. D of 28A will try out 2 sequence tables that were created from MANA called MANB and MANC. The latter is quite effective with delay hi. of 70 and delay lo. of 255 --- it is 4 wide and 21 tall and uses 2 for step size, because it is 2-bit moving animation. MANB is 1-bit moving (which is too small) and has 1 for step size since it's a 1-bit mover --- the same delays apply here. It's 3 wide and 21 tall. Printouts of MANB and MANC can be found near the front of Chapter 9C.

One of the reasons for doing a sequence diagram is that shapes must be saved into block-shapes at precise coordinates if they're to be effective. Another one is that keeping decent records of what you've done or are doing can be of enormous value. Another is that it's easier to do good sequences if you draw sketches first.

Look now at the last column. Why didn't 9 shapes get used when MANB and MANC were created from MANA? Well, MANA only works well in 6 of 28A because that program was especially designed for 3 moves per byte, or 2.33 bits per move. See how it was done:

PRINTOUT #33



```
]LIST5000-5230

5000 J = J + 1: IF J > 3 THEN J =
     1
5001 JT = J - 1: IF JT = 0 THEN J
     T = 3
5004  IF QW = 1 THEN Q = N + 1: GOTO
     5010
5005 Q = QW
5010  IF X = F THEN X% = X2 + ST(
     JT): GOTO 5100
5015 X% = X
5100  POKE 7,SQ(Q - 1): POKE 252,
     VT: POKE 253,VB: POKE 254,X%
      - ST(JT): POKE 255,X% - ST(
     JT) - WD: CALL 2116
5101  IF QL = 1 THEN  GOSUB 63010

5110 X = X + ST(J): IF X > E THEN
     X = F
5120 QW = QW + 1: IF QW > N THEN
     QW = 1
5130  POKE 7,SQ(QW): POKE 252,VT:
      POKE 253,VB: POKE 254,X: POKE
     255,X - WD: CALL 2116:X2 = X
     1:X1 = X
5200  FOR V = 1 TO Z(QW): NEXT
5230  RETURN
```

Line 5000 increments subscript for arrays in 5Ø1Ø and 511Ø. To understand this better, you should know that at line 3ØØ1 can be found:

ST(1)=1:ST(2)=Ø:ST(3)=Ø:J=2:X1=3:X2=3

Basically, the hor. block-shape coords. HL and HR get incremented every **third** drawing. This puts the program in sync. with the shapes (only feet and legs are shown) in the chart above.

The program (6 of 28A) is the same as the Applesoft one in Chapter 11B, except for the way it deals with incrementing by Ø two out of three times and by one the other time. If you don't see why we should be moving up the hor. coords. (HL and HR) every 3rd shape, then look at the 2nd to the last column in the chart. Sequence #4-5 are one byte ahead of #1-3 and #7-9 are 2 bytes ahead of #1-3. Notice how arrays make the "every 3rd shape" move very easy to signal/flag. Also notice that when shape sequences are run from Applesoft, arrays make it easy to handle whether or not the shape #s are mixed or consecutive. Assembly routines that run sequences in this graphics system are made to use consecutive shape #s. If you somehow have ended up with an out-of-order sequence of 7 shapes that need reordering, simply load the shape table into 4 of 28A and resave the right shapes into the right numbers. Or perform monitor memory moves (see page 59 of the white Apple Reference Manual) to interchange various shapes; Don't forget to use a temporary saving address not within the table for the interchanging, since once you've saved one shape on top of another shape, the latter gets lost.

Let's answer the earlier question: why didn't all 9 of MANA's shapes get used in MANB and MANC? Because MANB and MANC were created to work in our **general** (D of 28A) animation program which works best with 7-shape sequences. MANA's sequence works well only in 6 of 28A. It's not difficult to custom-build sequences to work in programs or custom-build programs to work with sequences, but the former is preferable. If you've forgotten, sequences of 7 work nice because there are 7 screen-visible bits per byte, and whether you move hor. coords. 1 byte between sequences because you're doing 1-bit moving or 2 bytes between sequences because you're doing 2-bit moving, 7 shapes is the correct # to keep things in sync. As you can readily imagine, 14 or 21 shapes can work fairly easily too (max. shapes in our tables is 23 as they're constructed).

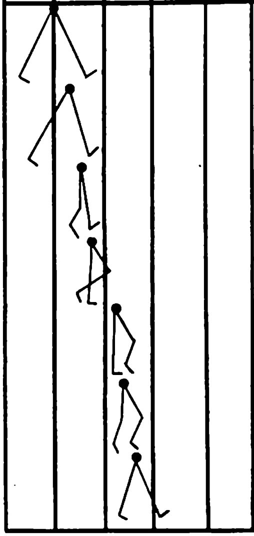
Early in 9C is a print-out of MANB and MANC. Let's look at how you'd create MANC from MANA, but first turn back and examine those 2 sequences. Notice again that the first sequence (MANB) is actually 4 bytes wide (top grid) but gets called 3 wide (lower grids). And MANC gets called 4 wide but is actually 5 wide. If you tell D of 28A that MANC is 5 wide it will scan for a 6-wide shape and goof up. Again --- I've mentioned this before --- think of it as the difference between inclusive and exclusive. In arithmetic if someone asks you how far it is from 28 to 24 or 24 to 28 you say 4. But from 24 **through** 28 is 5, because the word through indicates inclusively. So now if I tell you MANC is from byte 24 to 28 on the screen you'll say that it must therefore be 4 bytes wide. Four is the **correct** # to input and 4 is 28 minus 24, so --- yes --- **DO** say "4 bytes wide." But continue to know that the actual programs will, in such circumstances, scan bytes 24,25,26,27, and 28 --- which is 5 bytes of width. If you don't get this yet reread it. If you still don't get it, then all I can say is "don't blame Desenex."

Notice that in MANC shape 2 from MANA was used as sequence #2 and sequence #7. This will mean that shape #2 and #7 in MANC are identical --- confirm this visually with the print-out. This often happens in walking/running sequences.

You should also be aware that the MANC sequence involved 2-bit moves only. On the next page is the sequence diagram for MANC:

5-byte wide block shape

MANC



1st byte	2nd byte	3rd byte	4th byte	5th byte	seq. #	shape #	shape centered on hor. coord.:	VT,VB	move HL and HR up by 2?	hor. coords. of block-shape boundaries
					1	6	35	0,21	yes	28-63
					2	2	37	0,21	no	28-63
					3	7	39	0,21	no	28-63
					4	3	41	0,21	no	28-63
					5	8	43	0,21	no	28-63
					6	4	45	0,21	no	28-63
					7	2	47	0,21	no	28-63

This shape sequence would only have needed to be 4 wide (which you'd **call** 3) but I gave it extra for various experiments I was doing at the time. You're welcome to resave it without the extra space. It would speed it up a bit (which it doesn't need --- it already needs delay loops of 70 X 255=17,850 to slow it down).

If these shapes were going to be used (individually) in logical shift animation right-wards, they'd need the extra byte of space for shifting room, but it is only when similar shapes are needed in animation that logical shift works. It moves a shape over without changing it's shape. More on this later.

If you have any problem making shape sequences by loading them into 4 of 28A and PDL-defining them and later saving them, then make sure you actually DO the following --- we'll make a shape table (to be saved on your initialized data disk which you'll need sooner or later --- make sure it's the same DOS as the 28C (unlocked) disk), and we'll call it MAND, and it will be identical to MANC but 1 byte narrower. Here are the steps to follow:

1) Boot 28A and run 4, Scan Block-Shapes.

2) Use option 1 to load in vector shape table MAN, during which you can use 32768 for the address of the vector table, so that it won't clash with your block-shape addresses from $900 - $1FFF.

3) Load in shape #6, since it's the first shape in the MAND sequence you'll be doing. Give it X coord. of 35 and Y coord. of 12. Yes, the hor. coord. centered upon **is** going to be the vector shape's reference point's coord. Consult the sequence diagram for MANC.

4) When you're asked if you want another shape say yes ---there are 6 to go yet. Now put #2 at X,Y of 37,34. We'll move up X two dots per shape and move down Y twenty-two lines per shape so that they're not too close to one another. We should be able to PDL-define just like Jack The Giant Killer once all shapes are loaded. If I remember correctly, Jack The Giant Killer was known for "7-with-one-blow."

5) Next put #7 (see diagram) at 39,56; #3 at 41,78; #8 at 43,1ØØ; #4 at 45,122; and last but not least #2 at 47,144. Then choose no more shapes.

6) Choose option #7 now to PDL-define the block-shapes (which are still vector shapes at this point). You'll be using the game paddles to show the size of the block you desire. Read the instructions and continue. If the dot is invisible, fiddle with PDL#1 --- it's merely too low to appear.

7) Notice that X and Y are being monitored. Now move X to 29, Y to Ø and hit PDL button #Ø. Any # from 28 to 34 will start your block's X coord. at 28 and byte #4 (divide 28 by 7 to get the 4 --- there are 7 visible bits per byte). The PDL #Ø button will round off downwards to the nearest byte, the PDL#1 button will round off upwards to the nearest byte, so if you hit PDL#1 at X coord. 44, you'll have your block end just before hitting 49, which starts byte 7. Your shape would be 49-28 or 21 bits wide, which is 3 bytes in actuality, so you'd **call** it a width of 2 (see earlier in chapter).

8) Move to X=51, Y=21 and hit PDL#1. The 51 will give you an even 56 (divisible by 7) for right hor. coord. edge, so your shape will be 56-28=28 or 4 bytes wide (**call** it 3). This is the width we want. You'll see on the screen that the width is 3 and is from byte 4 to byte 7. This (and height of 21) you jot down along with the words MAND and "step 2" and "7 shapes,1-7." Remember, the program will be scanning not **to** byte 7 but **through** byte 7, all the way up to dot (hor.) 55 --- at the "edge" of dot 56, where the next byte starts. The program will scan bytes 4,5,6, and 7 for each vertical line in the shape --- but **continue to say width 3.**

9) When asked if the rectangle that now appears, defining the "block," is okay, say yes (hit Y and RETURN), and when asked if the program should scan this shape and PDL-define others too, say yes.

10) When asked for the block shape # of the shape you just defined, give 1 --- it'll be first in the sequence.

11) You'll be asked if you want the screen erased. Answer N for NO or you've wasted some work here!

12) Then you'll be asked for shape #. You already drew all the shapes you need so just do something like #1 at 99,99. The reason it's asking for more shapes (vector) is that normally you draw 1 shape in the upper left corner and then scan it, and then erase it (you were asked if you wanted to erase) and draw another one in its place and scan that --- etc. The advantage is that you may keep using the exact same Y coords. and merely increment X coords. The way we're **now** using this program you must actually add 22 to each set of Y coords. you use. The throwaway shape at 99,99 was just to keep the program going ---you'll be "throwing away" about 5 more of these.

13) You'll be asked if you want another shape, after doing the throwaway. Say no. You'll then go to the menu and choose #7 again.

14) Move the paddles until you're at 29,22 (X,Y) and hit #Ø; then move to 51,43 and hit #1 paddle. Say it's done okay, say you **want** to scan others, give "2" for block-shape #, say no for screen-erasing question, draw throwaway #1 at 99,99, and say no more shapes.

15) Choose option 7 in the menu and define sequence shapes #3-#7 by putting the dot at the following locations for PDL-defining:

For Block-Shape #:	Hit PDL #Ø when X,Y are at:	Hit PDL #1 when X,Y are at:
3	29, 44	51, 65
4	29, 66	51, 87
5	29, 88	51, 109
6	29, 110	51, 131
7	29, 132	51, 153

16) When you're asked if you want to scan the last shape say yes. You'll give it block-shape #7. Don't erase the screen. Draw the throwaway at 99,99, or wherever --- keep it out of the way. No more shapes now.

17) Choose option #9 to save PDL-defined shapes. When asked for the shape # you want all the data to go into, say #7. Give file name of MAND.

18) Next you'll be asked if you want to give # of shapes so that (# times 256) will be the table length, or if you'd rather let last shape # given (7) be last one in table, in which case the last byte of that shape will be the last one of the table. Choose option #2.

19) You'll see that your table took up 1624 bytes, which should be (256 *(last shape #-1)) + last shape length. Let's see if it is. 256X 6 is 1536. 1624-1536=88. Each of the #'s you give when inputing width and height for the above shapes is one short of the **real** width and height the computer uses, due to the inclusivity situation already carefully gone over. So the actual data array of this shape is 4 wide and 22 high, which gives an array-block of 4X22=88 bytes. All is well.

2Ø) It says give shape #7 1 place in the file --- which means #7 is no more than 256 bytes long. When you get back to the menu choose option 6 to load in MAND and then look at various shapes. Remember, however: MAND is **already** in memory. Seeing shapes now doesn't prove that MAND was saved correctly. To do that you need to reboot first.

21) Now go to the 28A MENU and choose D. When you're in D, give width 3, height 21, step size 2, right boundary 34, # of first shape --- 1, hi. delay byte 7Ø and low delay byte 255. He's a pretty smooth walker. If you desired for this animation to get even better, you'd turn MAND into MANE with doubling the # of shapes in the sequence to 14.

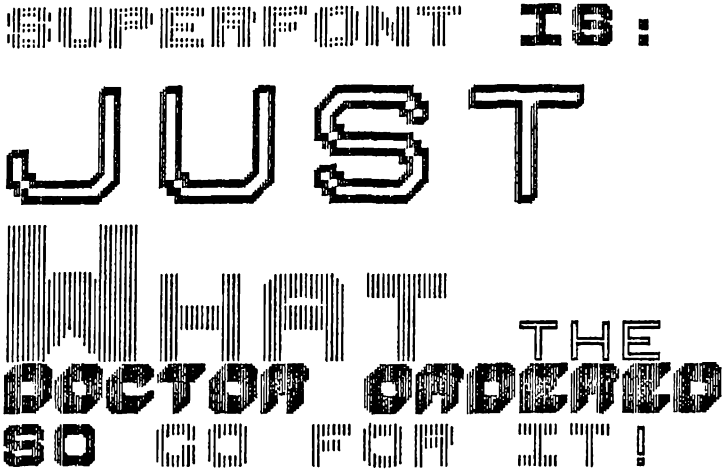
You'd then go for sequences of 14 with 1-bit movement that increments 2 bytes every 14 shapes. Only slight adjustments to TEST F (CALL36934) and TEST G (CALL36934) would be necessary to have 14 shapes per sequence. Remember that **colors** need 2-bit movements to work right. If you had a walking man who was moved 1-bit style, he'd flick back and forth between colors. You'd be better off with white men (oops, I sound like a Klansman). Another way to get smoother is to make the shapes larger. A colored man with 2-bit animation would be color-safe and very smooth if he was 2-3 times larger than our present shapes.

22) Don't be too close to the screen when viewing animation - you're supposed to see shapes, not "dots".

23) The reason that the arms are wiggly is that dots are either lined up horizontally, vertically, or at 45° angles. You must draw "wiggly" lines if you're trying to imitate 22° angles (for example).

12 SUPERFONT AND USING FONT

PRINTOUT #34



SUPERFONT IS:
JUST
WHAT THE
DOCTOR ORDERED
SO GO FOR IT!

12A. SUPERFONT

OPERATION:

A font is a little round device containing a full assortment of one size and style of printing type, as you probably know.

SUPERFONT is "super" because it contains 8 *styles* and 9 *sizes*, as well as sector-thrifty array-saved creations which are retrievable and which can be printed out. There is both manual and automatic scrolling, and manual and automatic saving. You get to choose from all 8 hi-res colors (2 whites, 2 blacks, violet, orange, green, and blue) and there's automatic carriage return and a full set of command options at the touch of a key. You can erase a letter, line or the entire screen. With a typewriter you use messy "white-out", but with a computer you hit "ESC", then B for black, then you backspace and type over the wrong characters. That's it.

USES:

There are several reasons why you might wish to use the "Superfont" program:

1) You might wish to write a program that starts out with some nice, big, sexy titles.

2) You might wish to compose a poster or sign using our "SUPERFONT", and hit CTRL P for a print-out, which you would then get Xeroxed at a copy shop.

3) You might invent programs that use "CHAR" in manipulating letters in interesting ways, like what happens in "Spirographing Letters" (a program found on our "Super Draw and Write" disk).

4) You can write titles to be used in graphic designing --- just cut them out after making print-outs.

5) Label things with cut-up print-outs. (You may have to adjust the print-out commands found in these programs --- printers vary a bit.)

6) You may wish to invent your own letters by inventing new STYLE formulae for the Superfont program (see lines 1ØØØ-1Ø2Ø).

7) We just got a letter the other day that was a combination of hi-res graphics and a font program, all combined artistically on a print-out. Think how impressed your friends will be if you write such letters!

There just isn't a faster way in the world to create a nice large-text graphic creation than to use "superfont" or other such font programs.

IMPORTANT NOTICE!!

When you want to save your screen creation, you must hit CTRL S **before** you start putting what you want saved on the screen. After you have finished your creation, hit CTRL F to complete the saving process. If you fill the screen after hitting CTRL S, your creation will be saved automatically.

12B. USING FONT

This program was originally written for a programmer who wanted to write a spelling program for Avant-Garde Creations. It's easy to use and play with and change.

All it does is prompt you to enter a # from 1-31 and then hit RETURN. The result is that you get a word written with our font characters, from the font shape table CHAR. If you want to change the color or style of the letters in the words, change line 559Ø (F can be 1-7 for style).

If you want to add words or change them, then add or change the literal strings in the data statements near the beginning of the program. All words need to be 14 letters or less in length.

This utility program gives you a good example of how you might like to manipulate font characters in an actual program you write.

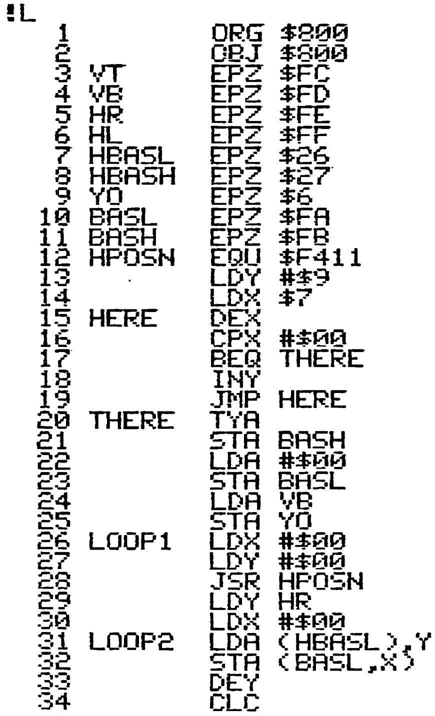
Whatever you need to do with words or titles or characters or numbers, you'll find it easy to do it with our font characters.

13 SHIFT ANIMATION

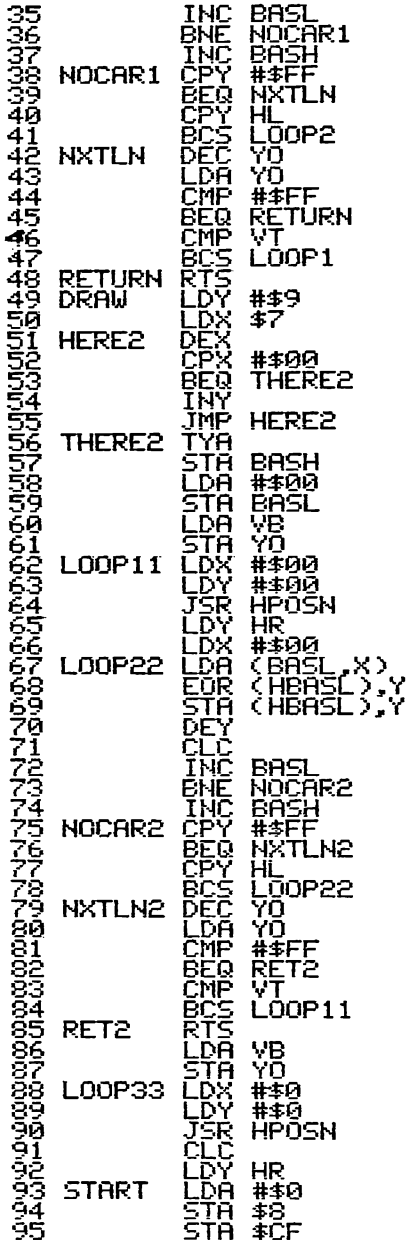
13A. ONE PAGE

We've already looked at how a shape may be shifted right, at the end of Chapter 9B. When ROL is used, the shape goes right, even though ROL is a left shift (technically, it's a rotation). The reason is that the bits in a hi-res byte show up in reverse order. By the same token, when ROR is used, the shape goes left. This is what happens in TEST I, which assembles the binary file TEST I (CALL2186). Let's check it out, remembering that if any details are unclear, a full explanation resides in the info-packed pages near the end of Chapter 9B.

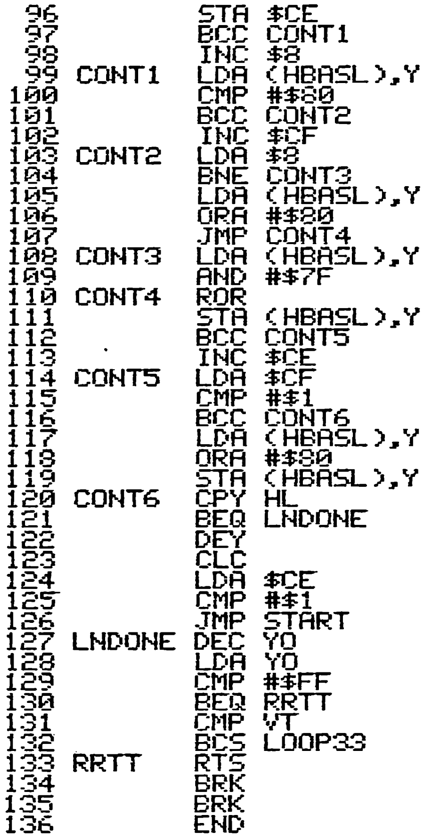
PRINTOUT #35



```
!L
   1              ORG  $800
   2              OBJ  $800
   3 VT           EPZ  $FC
   4 VB           EPZ  $FD
   5 HR           EPZ  $FE
   6 HL           EPZ  $FF
   7 HBASL        EPZ  $26
   8 HBASH        EPZ  $27
   9 YO           EPZ  $6
  10 BASL         EPZ  $FA
  11 BASH         EPZ  $FB
  12 HPOSN        EQU  $F411
  13              LDY  #$9
  14              LDX  $7
  15 HERE         DEX
  16              CPX  #$00
  17              BEQ  THERE
  18              INY
  19              JMP  HERE
  20 THERE        TYA
  21              STA  BASH
  22              LDA  #$00
  23              STA  BASL
  24              LDA  VB
  25              STA  YO
  26 LOOP1        LDX  #$00
  27              LDY  #$00
  28              JSR  HPOSN
  29              LDY  HR
  30              LDX  #$00
  31 LOOP2        LDA  (HBASL),Y
  32              STA  (BASL,X)
  33              DEY
  34              CLC
```



```
35              INC  BASL
36              BNE  NOCAR1
37              INC  BASH
38  NOCAR1      CPY  #$FF
39              BEQ  NXTLN
40              CPY  HL
41              BCS  LOOP2
42  NXTLN       DEC  YO
43              LDA  YO
44              CMP  #$FF
45              BEQ  RETURN
46              CMP  VT
47              BCS  LOOP1
48  RETURN      RTS
49  DRAW        LDY  #$9
50              LDX  $7
51  HERE2       DEX
52              CPX  #$00
53              BEQ  THERE2
54              INY
55              JMP  HERE2
56  THERE2      TYA
57              STA  BASH
58              LDA  #$00
59              STA  BASL
60              LDA  VB
61              STA  YO
62  LOOP11      LDX  #$00
63              LDY  #$00
64              JSR  HPOSN
65              LDY  HR
66              LDX  #$00
67  LOOP22      LDA  (BASL,X)
68              EOR  (HBASL),Y
69              STA  (HBASL),Y
70              DEY
71              CLC
72              INC  BASL
73              BNE  NOCAR2
74              INC  BASH
75  NOCAR2      CPY  #$FF
76              BEQ  NXTLN2
77              CPY  HL
78              BCS  LOOP22
79  NXTLN2      DEC  YO
80              LDA  YO
81              CMP  #$FF
82              BEQ  RET2
83              CMP  VT
84              BCS  LOOP11
85  RET2        RTS
86              LDA  VB
87              STA  YO
88  LOOP33      LDX  #$0
89              LDY  #$0
90              JSR  HPOSN
91              CLC
92              LDY  HR
93  START       LDA  #$0
94              STA  $8
95              STA  $CF
```



```
 96            STA  $CE
 97            BCC  CONT1
 98            INC  $8
 99  CONT1     LDA  (HBASL),Y
1ØØ            CMP  #$8Ø
1Ø1            BCC  CONT2
1Ø2            INC  $CF
1Ø3  CONT2     LDA  $8
1Ø4            BNE  CONT3
1Ø5            LDA  (HBASL),Y
1Ø6            ORA  #$8Ø
1Ø7            JMP  CONT4
1Ø8  CONT3     LDA  (HBASL),Y
1Ø9            AND  #$7F
11Ø  CONT4     ROR
111            STA  (HBASL),Y
112            BCC  CONT5
113            INC  $CE
114  CONT5     LDA  $CF
115            CMP  #$1
116            BCC  CONT6
117            LDA  (HBASL),Y
118            ORA  #$8Ø
119            STA  (HBASL),Y
12Ø  CONT6     CPY  HL
121            BEQ  LNDONE
122            DEY
123            CLC
124            LDA  $CE
125            CMP  #$1
126            JMP  START
127  LNDONE    DEC  YO
128            LDA  YO
129            CMP  #$FF
13Ø            BEQ  RRTT
131            CMP  VT
132            BCS  LOOP33
133  RRTT      RTS
134            BRK
135            BRK
136            END
```

The biggest difference between TEST I, above, and TEST H, which was handled in the end of Chapter 9B, is that we're shifting shapes leftwards in TEST I, but rightwards in TEST H. A "bit" different procedure is required:

From line 1-85 are the scanning and drawing programs.

The programs start the same. **Line 86-87** puts the vert. bottom block-shape coordinate into both $6 and the accumulator.

Line 88-9Ø puts Ø in X and Y registers, so now the HPOSN routine has all 3 registers properly loaded so it is run. (HBASL and HBASH will be loaded with the VB/byte #Ø hi-res address. This address will be indexed by Y, which gets loaded with HR in line 92).

Line 91 clear carry so line 97 works okay.

Line 92 stick Y into HR so indexing works with (HBASL),Y.

Line 93-96 zero the 3 flags:

$8	bit Ø flag, last cycle
$CE	bit Ø flag, this cycle
$CF	bit 7 flag

Line 97 if bit 0 was a 0 after last ROR, then there's no "dot" to move over from that byte to this byte, so carry is clear and BCC detects this.

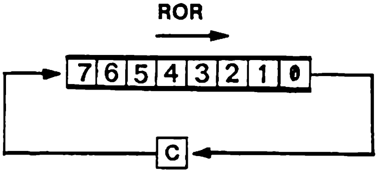
Line 98 set bit 0 flag, last cycle; bit 0 in last cycle was set, and during ROR it set carry, so we want to stick the bit into bit 7 temporarily in **this** ROR cycle, so that when ROR happens bit 7 will go down to bit 6. The net result is dumping last cycle's 0 bit into this cycle's 6 bit, which is the (visible) 1-bit lefward shift we're after.

Line 92-102 check to see if the color bit is set; if it is, set the bit 7 flag in $CF; $80 is 128, which requires bit 7 set.

Line 103-106 check to see if bit 0 last cycle flag is set; if it is then set flag 7 now so that ROR will put that "dot" into the first visible rightmost bit of this byte, #6. ORA #$80 means set bit 7. See last part of Chapter 9B.

Line 107-109 if bit 0 last cycle flag was NOT set you'll need to put a 0 into bit 7 so that bit 6 will be 0 after ROR. To refresh you on AND #$7F, it simply sets bit 7 to 0. The way it does it is by saying that the accumulator bit and data (#$7F) bits that get compared will result in an ON bit **only** if **both** compared bits are 1. Since $7F is 01111111, then bits 1-6 are left as is, but bit 7 is definitely going to end up a 0. Line 107 avoids lines 108-109 if bit 7 was just set to 1 purposely in lines 105-106. Line 108-109 sets bit 7 to 0.

Line 110-111 performs an ROR instruction on the present byte and stores it in the screen byte it came from. An ROR is an LSR that puts the carry bit rather than a 0 into the vacated 7 bit. An LSR is called a logical shift operation. An ROR is called a 9-bit rotation instruction. An ROR is a shift, but an LSR is NOT a rotation. An ROR happens to be a shift that includes the rotational aspect of using the carry bit rather than a 0 to fill the vacated 7 bit. As we've said before, I'd have called all this stuff **rotation animation** rather than **shift animation** except for the fact that this would sound misleading --- like a way to get shapes to rotate, which it **isn't**.



Line 112 if 0 bit that gets rotated into carry (C) status flag then carry is clear and BCC detects this and sends us to line 114.

Line 113 if carry was set because the 0 bit was 1 before ROR, then increment the "0 flag this cycle" flag. We'll need to use this flag to determine how to set the carry just before going on to the next byte.

Line 114-116 check the bit 7 flag (color bit flag) --- if it's clear then go on to line 120.

Line 117-119 if bit 7 flag was set, stick a 1 back into bit 7. See Chapter 9B if confused.

Line 120-121 see if we're done with the current line; if so, go to 127 and move up a line.

Line 122 move left one byte before next rotation (ROR) cycle begins.

Line 123 clear carry so that line 125 will properly set carry status, even though it isn't needed, since the CMP instruction conditions the carry status flag correctly even if this line is a SEC instruction. I guess I used this instruction to remind myself what I was doing! (Poor excuse --- let's face it, DON, you couldn't Budge it, so you Fudged it!) I probably used this unnecessarily quite a few times in this package. Oh well, it won't hurt anything. Anyway, after CMP instruction BCC detects the accumulator being less than the data, and BCS detects the accumulator being equal to or greater than the data.

Line 124-125 see if Ø bit was 1 before latest ROR; if so, then set carry with the CMP instruction since accumulator is equal to the data.

Line 126 now go to line 93 to rotate the next byte leftwards.

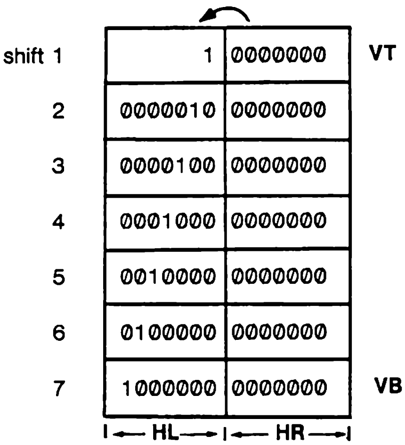
Line 127-128 if we're here, we've skipped the conditioning of the carry flag relative to the status of the "Ø bit this cycle" flag, since there's no byte left of our present position in this block-shape to receive the "dot" we'd be shifting left. Let's take a closer look at this: Since we're not putting any more dots in bytes to the left of us, it means it's time to move up to the next higher vertical line of bytes. We **don't** want a conditioned carry for this, but a zeroed carry, which will in line 91 once line 132 sends us back up for more cycling on a fresh line. If the carry was set, and we went up a line and left it set, and then started rotating bytes at HR, then our shape's right-most dot would stay where it was, not shift left, in appearance.

One thing to notice here is that if we were to have a shape inside a block-shape whose actual shape-dots were closer than 7 dots (1 byte) from the left edge of the block-shape, then we'd be shifting dots off into never-never-land, in effect, since the rotations must all take place within the current boundaries of the block shape. So in a block-shape that's going to be shifted left with ROR (**the screen's left** is the direction dots move during ROR), then the entire left-most column of bytes of that block-shape need to be blank, off, zero. The right-most column of bytes need not be blanks.

For rightward shape-shifting with ROL the opposite is true -- define (with PDLs) the block-shape so that the rightmost column of bytes in the block-shape are as bare as Old Mother Hubbard's Cupboard --- not even a bone is allowed. The leftmost byte column may have **on** bits. If you'll be moving both ways via shift-animation, then leave a bare-byte column on **both** sides, or you'll soon be watching shapes distort --- or at least go on some unplanned crash diets. This is one of the tricks to learn about shift animation. If you forget it, don't blame Desenex **or** Don Fudge.

If you're still wondering **why** shapes would goof up if not for the bare byte column in the direction of shift, think of it like this:

If done right, 7 shifts of a "1-dot shape" would do the following:



shift 1	1	0000000	VT
2	0000010	0000000	
3	0000100	0000000	
4	0001000	0000000	
5	0010000	0000000	
6	0100000	0000000	
7	1000000	0000000	VB
	I←— HL—→I	I←— HR—→I	

If done wrong, the exact same thing would happen, but HL would be put where HR is and HL would be the same as HR. (If the shape was wider then HR could be several bytes to the right, it wouldn't matter ---the main thing here is that HL would be one byte further right than it should be for shift-animation).

Think of HL being the byte # of the left-most column of bytes, and think of these HL bytes as composed of all 0's, during (ROR) leftward shape-shifting. Once 7 shifts have occured, some or all of these HL bytes will have some turned on bits.

In the case where HL was moved to HR, the block-shape's definition wouldn't include the dots that got rotate-shifted beyond the HL byte, so line 120 and 121 would see to it that we simply went up to the next line higher once we'd shifted HL bits into left-ward oblivion ---there'd be no place in the routine that would turn these shifted dots into visible shape dots left of the HL byte. So they'd simply disappear. You may **want** this for an effect someday. But start out your shifts now in ways that preserve shape integrity.

So all we do in lines 127-128 is move up to the next line by decrementing $6, the current line holder.

Line 129-132 checks to see if we're either trying to go higher on the screen than a Y coord. of 0 or of VT --- either is a no-no. If we **are** attempting such silly things we're RTSed out of the routine --- the shape has been shifted over 1 dot.

This should all but clear up any shape-shifting questions relative to a one-dot shit. But now let's look at why we've mentioned shifting 7 times. To some of you it's obvious. But I won't make the error that Call Apple and Apple Orchard often make --- assuming the readers know more than they do; those are 2 great mags., but they often need more background, context, or additional info. about how to **use** the ideas or data they print.

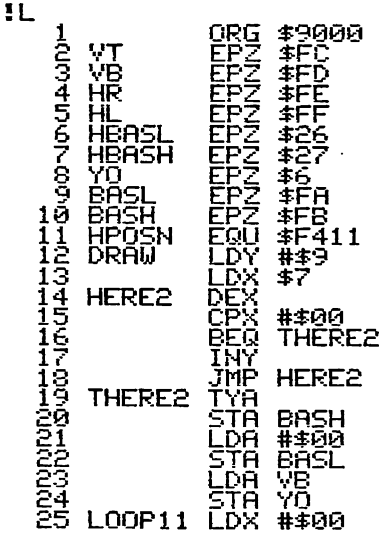
Once you've shifted 7 times, what do you suppose happens? Reagan gives his $ to the poor? No. California falls into the sea but jumps back out again because the water's too cold? No. Give up?

The answer is: it becomes HL-and-HR-changing time. (Not very dramatic, but quite true.) If you're shape-shifting leftwards, you'd decrement the hor. right byte coord. and the hor. left byte coord. every seven shifts. You wouldn't redraw the shape. All you'd do is say: "Okay, I acknowledge that the old grey shape she just ain't where she used to be." Physically on the screen you've got a shape that's 7 dots left of where it started out. If you were to keep on shifting without telling the routine about your changed block-shape position, the routine would begin throwing its left-most dots off the cliff of infinity and into the Black Hole. Avoid this catastrophe. Remember to decrement or increment HR and HL (depending upon shift direction). You'll find that the following routines **will** remember. (See next chapter.)

13B. TWO-PAGE SHIFT ANIMATION

TEST J will do shift-animation to the right. It's 2-page flipping animation and it's colorsafe. No, I don't mean laundry detergent! I mean it shifts twice before flipping pages. The result is that colors have no problem --- we're doing the old 2-bit shift. The program that drives TEST J (CALL36934) or TEST K (CALL36934) is 3 of 28B. Like all other programs in the system, it's listable via E or F of 28B. TEST K is like TEST J except it animates leftwards. But let's see TEST J first:

PRINTOUT #36



```
!L
  1         ORG $9000
  2 VT      EPZ $FC
  3 VB      EPZ $FD
  4 HR      EPZ $FE
  5 HL      EPZ $FF
  6 HBASL   EPZ $26
  7 HBASH   EPZ $27
  8 YO      EPZ $6
  9 BASL    EPZ $FA
 10 BASH    EPZ $FB
 11 HPOSN   EQU $F411
 12 DRAW    LDY #$9
 13         LDX $7
 14 HERE2   DEX
 15         CPX #$00
 16         BEQ THERE2
 17         INY
 18         JMP HERE2
 19 THERE2  TYA
 20         STA BASH
 21         LDA #$00
 22         STA BASL
 23         LDA VB
 24         STA YO
 25 LOOP11  LDX #$00
```

```
26         LDY  #$00
27         JSR  HPOSN
28         LDY  HR
29         LDX  #$00
30 LOOP22  LDA  (BASL,X)
31         EOR  (HBASL),Y
32         STA  (HBASL),Y
33         DEY
34         CLC
35         INC  BASL
36         BNE  NOCAR2
37         INC  BASH
38 NOCAR2  CPY  #$FF
39         BEQ  NXTLN2
40         CPY  HL
41         BCS  LOOP22
42 NXTLN2  DEC  YO
43         LDA  YO
44         CMP  #$FF
45         BEQ  RET2
46         CMP  VT
47         BCS  LOOP11
48 RET2    RTS
49 STT     JSR  $F3E2
50         LDA  #$0
51         STA  $9
52         STA  $C052
53         JSR  $F3D8
54         LDA  #$40
55         STA  $E6
56         LDA  #$0
57         STA  $C054
58         JSR  DRAW
59         JSR  SHIFT
60         JSR  SHIFT
61         LDA  #$0
62         STA  $C055
63         LDA  #$20
64         STA  $E6
65         JSR  DRAW
66         INC  $9
67         INC  $9
68 LOOP    LDA  #$20
69         STA  $E6
70         LDA  #$0
71         STA  $C055
72         JSR  SHIFT
73         JSR  SHIFT
74         LDX  #$2
75         INC  $9
76         LDA  $9
77         CMP  #$8
78         BEQ  SUB10
79 CONT10  JSR  SHIFT
80         JSR  SHIFT
81         LDA  #$40
82         STA  $E6
83         LDA  #$0
84         STA  $C054
85         JSR  SHIFT
86         JSR  SHIFT
```

```
       LDX #$1
       INC $9
       LDA $9
       CMP #$8
       BEQ SUB10
CONT20 JSR SHIFT
       JSR SHIFT
       LDA HR
       CMP #$24
       BEQ RTN
       BNE LOOP
RTN    RTS
SUB10  INC HL
       INC HL
       INC HR
       INC HR
       LDA #$1
       STA $9
       CPX #$2
       BEQ CONT10
       BNE CONT20
SHIFT  LDA YB
       STA YO
LOOP33 LDX #$0
       LDY #$0
       JSR HPOSN
       CLC
       LDY HL
START  LDA #$0
       STA $8
       STA $CF
SHFT   LDA (HBASL),Y
       ROL
       STA (HBASL),Y
       BCS SUB1
       BCC CONT1
SUB1   INC $8
CONT1  CMP #$80
       BCS SET64
       BCC CONT2
SET64  INC $CF
CONT2  LDA $8
       BNE SUB2
       LDA (HBASL),Y
       AND #$7F
       STA (HBASL),Y
       JMP SUB3
SUB2   LDA (HBASL),Y
       ORA #$80
       STA (HBASL),Y
SUB3   CPY HR
       BEQ LNDONE
       INY
       CLC
       LDA $CF
       CMP #$1
       JMP START
LNDONE DEC YO
       LDA YO
       CMP #$FF
       BEQ RRTT
```

```
148          CMP VT
149          BCS LOOP33
15Ø RRTT     RTS
151          BRK
152          BRK
153          END
```

Let's ignore lines 1-48, since that's merely the block-shape drawing routine. Let's also ignore lines 1Ø8-153, since these are the exact same shape-shifting lines we've already gone carefully through in Chapter 9B. That leaves 49-1Ø7.

Line 49 HGR.

Line 50-51 zeroes shape-counter.

Line 52 page 1 full-page graphics.

Line 53 HGR2.

Line 54-55 draw on page 2.

Line 56-57 display page 1.

Line 58 normal block-shape draw.

Line 59-60 shift it right twice.

Line 61-62 display page 2.

Line 63-64 draw on page 1.

Line 65 normal block-shape draw.

Line 66-67 you've drawn your 1st and 2nd shapes, one on each page, so increment the shape counter ($9).

Line 68-69 draw on page 1.

Line 70-71 display page 2.

Line 72-73 double shift.

Line 74 by putting 2 into X you signal you're in the LOOP section.

Line 75-78 increment shape counter and compare it to 8. If it's 8 go to SUB1Ø routine, since it's time to reset counter to 1 and increment HL and HR.

Line 79-80 double shift.

Line 81-82 draw on page 2.

Line 83-84 display page 1.

Line 85-86 display page 2.

Line 87 by putting 1 into X you signal you're in the CONT1Ø section.

Line 88-91 increment shape counter and compare it to 8. If it's 8 go to SUB1Ø routine, since it's time to reset counter to 1 and increment HL and HR.

Line 92-93 double shift.

Line 94-98 check HR to see if we're at 36 yet. If we are, quit entire routine. If not, go back to LOOP.

Line 99-102 increment HL and HR twice. With 2-bit moving you need to have a step-size of 2 for hor. coords.

Line 103-104 reset shape counter to 1.

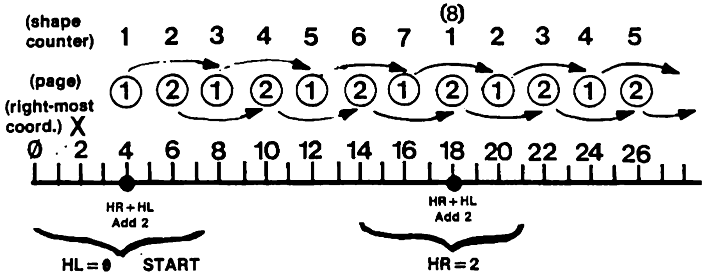
Line 105-106 if you just came from LOOP, go to CONT1Ø.

Line 107 if you just came from CONT1Ø go to CONT2Ø.

The first thing to remember about a routine that does 7 two-bit shifts is that 7X2=14. This means that we need **two** blank byte-columns included in the block-shape's definition, and they must be on the right of the shape. One column of blank bytes is only good for 1-bit shift-animation.

You're probably wondering about TEST J's weird construction. In LOOP and CONT2Ø are 2 shifts, but in CONT1Ø are 4 shifts. What's it all about, Alfie?

Well, once we've double-shifted 7 times (notice you never have to go back and erase? --- that's the best part of shift-moving) we need to have the shape counter send us to SUB1Ø to reset counter and double-increment HR and HL (step size is 2 bytes per step). But remember that each hi-res page shifts 4 times, and this is because the first 2 shifts bring you up to the other hi-res page's coordinates, while the second 2 shifts bring it 2 past the other page. Look at this (X will mean coord. of right-most ON shape-dot):



In **lines 49-67** we get 2 shapes drawn, one at right-most X coord. of 4 and one shifted 2 bits over and at 6. **Both** have HR=2 and HL=Ø, the first byte column on the screen. The counter now says 2. (X will continue to mean coord. of right-most **ON** shape dot.

In **lines 68-8Ø** we shift the shape at X=4 over to X=6 and check to see if incrementing the counter gets it to 8, which means it's time to raise HL and HR. It's not so we continue shifting page one's shape to X=8.

In **lines 81-93** we switch to page 2 and shift its shape to 8, check counter and find it's only 4 so we finish shifting it over to 1Ø. We're sent back to LOOP.

Back in **lines 68-8Ø** on page one now we shift it from 8 to 1Ø, check counter --- it's at 5, shift from 1Ø to 12.

Now in **lines 81-93** on page two we shift the shape from 1Ø to 12, counter's only at 6, so we shift from 12 to 14.

Next in **lines 68-8Ø** on page one we shift from 12 to 14, check counter --- it's only 7, so we shift from 14 to 16.

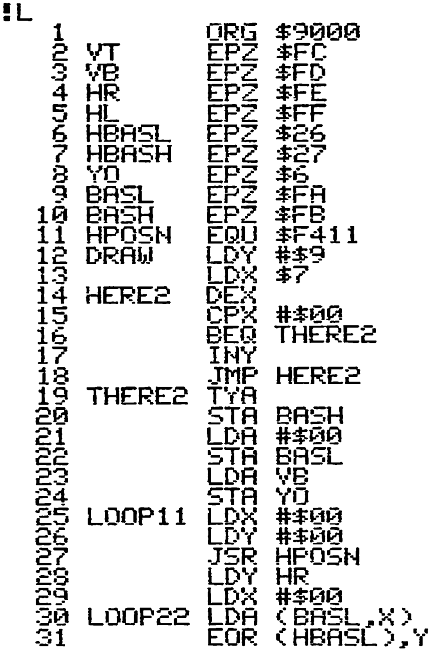
Then in lines 81-93 on page two we shift from 14 to 16 and check the counter. It's at 8! We go to SUB1Ø and increment HL and HR twice and shift 16 into 18, and reset the counter at 1. When HL was Ø and HR was 2 we were dealing with dots (hor.) of Ø to 2Ø. But now we're dealing with dots 14 to 34. Notice the right-most shape-dot's coords. (**not** byte #) is again 4 dots from the left edge of the block-shape. Let's say the shape dots started in X=0 and went up to X=4 before we incremented HR + HL by 2 bytes. Now the shape dots start at 14 and go to 18. Notice how after each HR and HL double-incrementation, there are always at least 2 blank bytes to the right block-shape edge, so that the 2 columns of blank bytes give us the 14 bits of shifting room we require (2X7). In our case there are 2 bytes + 2 bits.

In the above example what was meant by "shifting 8 into 10" was actually this:

The entire shape from dot 0 to dot 20 was shifted within itself so that all visible dots rotated over rightwards 2 dots. The actual visible dots in this block-shape range from 4 to 8 (5 in a row) at this time, even though they started out at a range of from 0 to 4. We are double-shifting all bytes in the entire shape and the **visible** effect will be that the dots which are now from 4 to 8 will range, after the rotation-shift, from 6 to 10. As you remember, the "X" referred to in the above shifting description refers to the visible shape's right-most dot's coordinate. Don't forget this or I'll send a flock of drunk pigeons (who've just gotten back from a heavy meal in an out-of-the-way part of Mexico) to roost on your freshly waxed T-Bird overnight. And that ain't no Fudge.

Let's see what TEST K has in store for us. Its binary file is TEST K (CALL36934) and its program is also 3 of 28B. It will move a shape **leftwards** by shift animation as opposed to rightwards like TEST J. Here's TEST K:

PRINTOUT #37



```
!L
           ORG  $9000
  VT       EPZ  $FC
  VB       EPZ  $FD
  HR       EPZ  $FE
  HL       EPZ  $FF
  HBASL    EPZ  $26
  HBASH    EPZ  $27
  YO       EPZ  $6
  BASL     EPZ  $FA
  BASH     EPZ  $FB
  HPOSN    EQU  $F411
  DRAW     LDY  #$9
           LDX  $7
  HERE2    DEX
           CPX  #$00
           BEQ  THERE2
           INY
           JMP  HERE2
  THERE2   TYA
           STA  BASH
           LDA  #$00
           STA  BASL
           LDA  VB
           STA  YO
  LOOP11   LDX  #$00
           LDY  #$00
           JSR  HPOSN
           LDY  HR
           LDX  #$00
  LOOP22   LDA  (BASL,X)
           EOR  (HBASL),Y
```

```
               STA (HBASL),Y
               DEY
               CLC
               INC BASL
               BNE NOCAR2
               INC BASH
        NOCAR2 CPY #$FF
               BEQ NXTLN2
               CPY HL
               BCS LOOP22
        NXTLN2 DEC YO
               LDA YO
               CMP #$FF
               BEQ RET2
               CMP VT
               BCS LOOP11
        RET2   RTS
        STT    JSR $F3E2
               LDA #$0
               STA $9
               STA $C052
               JSR $F3D8
               LDA #$40
               STA $E6
               LDA #$0
               STA $C054
               JSR DRAW
               JSR SHIFT
               JSR SHIFT
               LDA #$0
               STA $C055
               LDA #$20
               STA $E6
               JSR DRAW
               INC $9
               INC $9
        LOOP   LDA #$20
               STA $E6
               LDA #$0
               STA $C055
               JSR SHIFT
               JSR SHIFT
               LDX #$2
               INC $9
               LDA $9
               CMP #$8
               BEQ SUB10
        CONT10 JSR SHIFT
               JSR SHIFT
               LDA #$40
               STA $E6
               LDA #$0
               STA $C054
               JSR SHIFT
               JSR SHIFT
               LDX #$1
               INC $9
               LDA $9
               CMP #$8
               BEQ SUB10
        CONT20 JSR SHIFT
        JSR  SHIFT
        LDA  HL
        CMP  #$3
        BEQ  RTN
        BNE  LOOP
RTN     RTS
SUB10   DEC  HL
        DEC  HL
        DEC  HR
        DEC  HR
        LDA  #$1
        STA  $9
        CPX  #$2
        BEQ  CONT10
        BNE  CONT20
SHIFT   LDA  VB
        STA  YO
LOOP33  LDX  #$0
        LDY  #$0
        JSR  HPOSN
        CLC
        LDY  HR
START   LDA  #$0
        STA  $8
        STA  $CF
        STA  $CE
        BCC  CONT1
        INC  $8
CONT1   LDA  (HBASL),Y
        CMP  #$80
        BCC  CONT2
        INC  $CF
CONT2   LDA  $8
        BEQ  CONT3
        LDA  (HBASL),Y
        ORA  #$80
        JMP  CONT4
CONT3   LDA  (HBASL),Y
        AND  #$7F
CONT4   ROR
        STA  (HBASL),Y
        BCC  CONT5
        INC  $CE
CONT5   LDA  $CF
        CMP  #$1
        BCC  CONT6
        LDA  (HBASL),Y
        ORA  #$80
        STA  (HBASL),Y
CONT6   CPY  HL
        BEQ  LNDONE
        DEY
        CLC
        LDA  $CE
        CMP  #$1
        JMP  START
LNDONE  DEC  YO
        LDA  YO
        CMP  #$FF
        BEQ  RRTT
        CMP  VT
```

```
154          BCS LOOP33
155 RRTT     RTS
156          BRK
157          BRK
158          END
```

Let's just look at the difference between this routine and TEST J. First, its shift-rotation routine is the leftward one first seen in TEST I. That's from line 108-158.

The 1st difference is in **line 95** --- since we're moving left the limit # is 3, not 36.

The 2nd difference is in line **99-102** --- we must double-decrement HR and HL every 7 shapes, not double-increment.

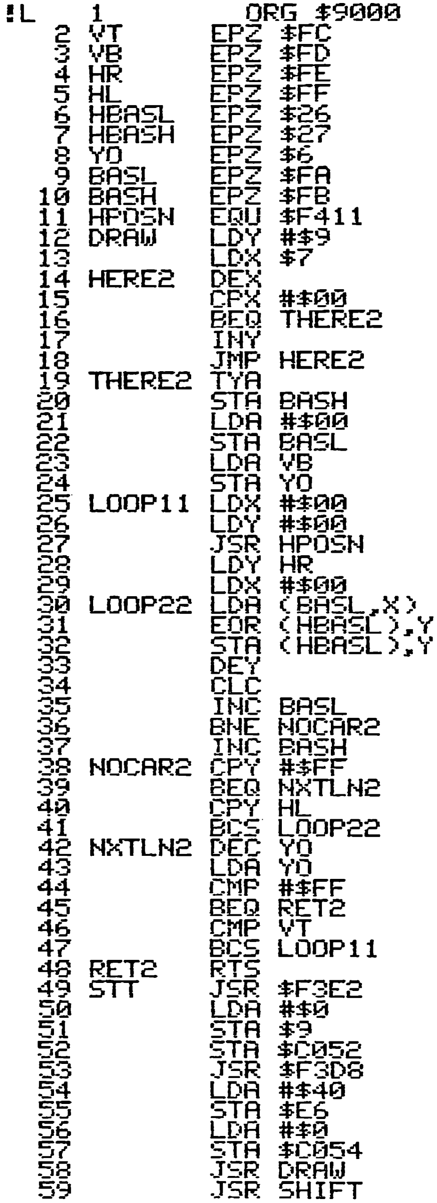
The 3rd and final difference is in the shape to be shifted. It obviously needs 2 blank columns of bytes within its block-shape definition, column that will be the first 2 left byte columns in the shape. No ON dots allowed. The ON dots may start anywhere after the 14th horizontal block-shape dot.

Next we'll look at this last routine (TEST K) in action. Run 3 of 28B and say you want to move left with shape table MANDG using shape #1 and a VT of 0, a VB of 39, an HR of 39, and an HL of 33. The shape is 40 high by 7 wide in actuality, so we'll call it 39X6. That's why we gave it coords. such that VB-VT=39 and HR-HL=6.

Now why is the shape 280 bytes (40X 7) when it could obviously be smaller, even 22X3=66? Simple, MANDG is a test shape. It has at least 8 extra blank dots above and below it and 2 blank bytes on either side of it (I'm speaking of the visible part of the shape now). It can move in any direction without losing any part of its shape due to running out of internal shifting room. So what **is** the smallest block-shape of the same man shape that would work to try out leftward shift animation in 3 of 28B? Simple, take 22X3 and add 2 to the width and you'll then get 22X5=110 bytes in the block-shape data-array, which you'd **call** 21X4. See previous chapters if this sounds strange --- it's all explained. This extra width of 2 blank bytes on the left side of the visible shape will give shifting room for colorsafe 2-bit animation. Since only 40% as many bytes will be shifted per shift cycle, the shape will be 2 1/2 times as quick at shifting leftwards as it was when it was 280 bytes large. This will be fortunate, since it was going a bit slow to be good animation before. Now you should be getting some idea of the maximum size a block-shape needs to be to be a good rotation-shift animation candidate.

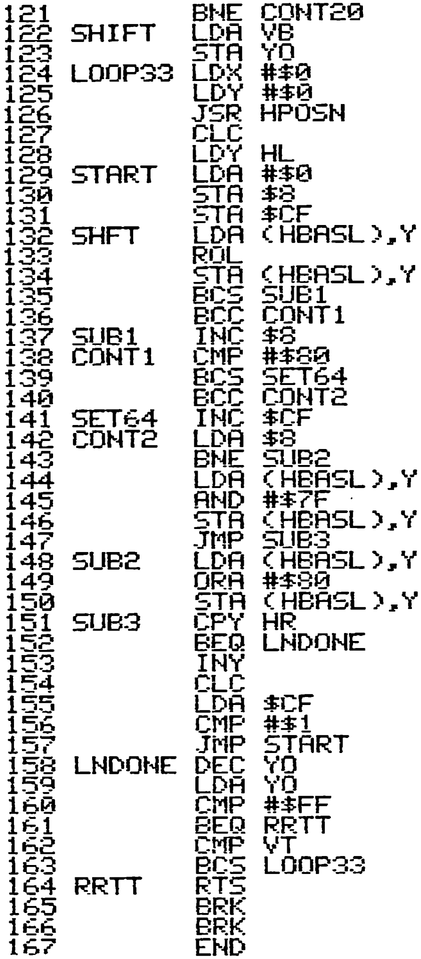
Now let's look at TEST M (found in 4 of 28B), which is quite like TEST J except it shifts shapes up or down **in addition to** rotation-shifting them --- the net result is diagonal animation, slower, of course than **horizontal** animation. TEST M creates a binary file TEST M (CALL36934) which moves things rightwards and either up or down:

PRINTOUT #38



```
!L     1          ORG $9000
       2 VT      EPZ $FC
       3 VB      EPZ $FD
       4 HR      EPZ $FE
       5 HL      EPZ $FF
       6 HBASL   EPZ $26
       7 HBASH   EPZ $27
       8 YO      EPZ $6
       9 BASL    EPZ $FA
      10 BASH    EPZ $FB
      11 HPOSN   EQU $F411
      12 DRAW    LDY #$9
      13         LDX $7
      14 HERE2   DEX
      15         CPX #$00
      16         BEQ THERE2
      17         INY
      18         JMP HERE2
      19 THERE2  TYA
      20         STA BASH
      21         LDA #$00
      22         STA BASL
      23         LDA VB
      24         STA YO
      25 LOOP11  LDX #$00
      26         LDY #$00
      27         JSR HPOSN
      28         LDY HR
      29         LDX #$00
      30 LOOP22  LDA (BASL,X)
      31         EOR (HBASL),Y
      32         STA (HBASL),Y
      33         DEY
      34         CLC
      35         INC BASL
      36         BNE NOCAR2
      37         INC BASH
      38 NOCAR2  CPY #$FF
      39         BEQ NXTLN2
      40         CPY HL
      41         BCS LOOP22
      42 NXTLN2  DEC YO
      43         LDA YO
      44         CMP #$FF
      45         BEQ RET2
      46         CMP VT
      47         BCS LOOP11
      48 RET2    RTS
      49 STT     JSR $F3E2
      50         LDA #$0
      51         STA $9
      52         STA $C052
      53         JSR $F3D8
      54         LDA #$40
      55         STA $E6
      56         LDA #$0
      57         STA $C054
      58         JSR DRAW
      59         JSR SHIFT
```

```
 60        JSR SHIFT
 61        JSR $800
 62        LDA #$0
 63        STA $C055
 64        LDA #$20
 65        STA $E6
 66        JSR DRAW
 67        INC $9
 68        INC $9
 69 LOOP   LDA #$20
 70        STA $E6
 71        LDA #$0
 72        STA $C055
 73        JSR SHIFT
 74        JSR SHIFT
 75        JSR $800
 76        LDX #$2
 77        INC $9
 78        LDA $9
 79        CMP #$8
 80        BEQ SUB10
 81 CONT10 JSR SHIFT
 82        JSR SHIFT
 83        JSR $800
 84        LDA #$40
 85        STA $E6
 86        LDA #$0
 87        STA $C054
 88        JSR SHIFT
 89        JSR SHIFT
 90        JSR $800
 91        LDX #$1
 92        INC $9
 93        LDA $9
 94        CMP #$8
 95        BEQ SUB10
 96 CONT20 JSR SHIFT
 97        JSR SHIFT
 98        JSR $800
 99        LDA HR
100        CMP #$24
101        BEQ RTN
102        LDA VB
103        CMP #$BF
104        BEQ RTN
105        BNE LOOP
106 RTN    RTS
107 SUB10  INC HL
108        INC HL
109        INC HR
110        INC HR
111 XYZ    LDA #$7
112        STA $CE
113        INC VT
114        INC VB
115        DEC $CE
116        BNE XYZ
117        LDA #$1
118        STA $9
119        CPX #$2
120        BEQ CONT10
```



```
121           BNE CONT2Ø
122 SHIFT     LDA VB
123           STA YO
124 LOOP33    LDX #$Ø
125           LDY #$Ø
126           JSR HPOSN
127           CLC
128           LDY HL
129 START     LDA #$Ø
13Ø           STA $8
131           STA $CF
132 SHFT      LDA (HBASL),Y
133           ROL
134           STA (HBASL),Y
135           BCS SUB1
136           BCC CONT1
137 SUB1      INC $8
138 CONT1     CMP #$8Ø
139           BCS SET64
14Ø           BCC CONT2
141 SET64     INC $CF
142 CONT2     LDA $8
143           BNE SUB2
144           LDA (HBASL),Y
145           AND #$7F
146           STA (HBASL),Y
147           JMP SUB3
148 SUB2      LDA (HBASL),Y
149           ORA #$8Ø
15Ø           STA (HBASL),Y
151 SUB3      CPY HR
152           BEQ LNDONE
153           INY
154           CLC
155           LDA $CF
156           CMP #$1
157           JMP START
158 LNDONE    DEC YO
159           LDA YO
16Ø           CMP #$FF
161           BEQ RRTT
162           CMP VT
163           BCS LOOP33
164 RRTT      RTS
165           BRK
166           BRK
167           END
```

Let's look at the difference from TEST J:

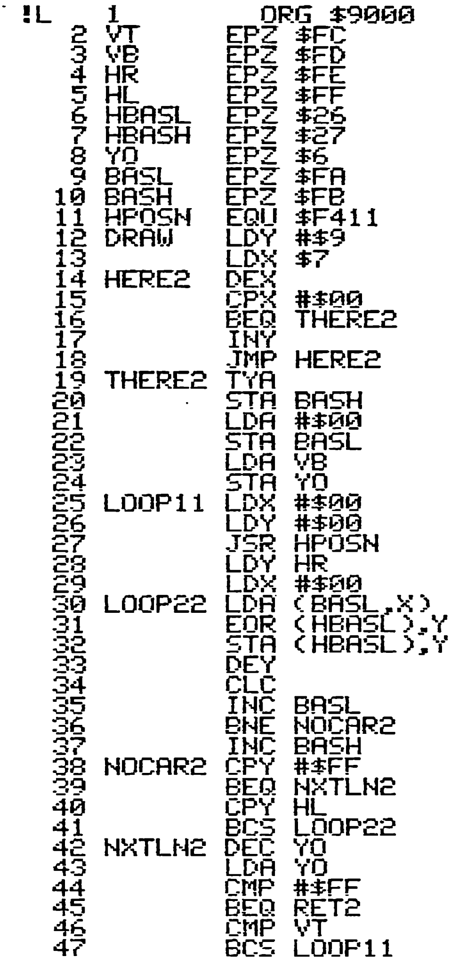
The drawing program from 1-48 is the same. The STT starting routine from 49-68 is similar, but not the same. At line 61 there is a JSR $8ØØ. This is the vertical movement routine. More on that later. The LOOP section is the same except for line 75 which is again JSR $8ØØ. The CONT1Ø section is the same except for JSR $8ØØ at 83 and 9Ø. CONT2Ø has a JSR $8ØØ at line 98.

Line 99-1Ø6 checks not only horizontal limits, but vertical limit of 191.

Line 1Ø7-121 is the SUB1Ø section, and it has a few differences. Line 111-112 sticks a 7 into $CE, which is the vert. change counter. It makes sure you move the VB and VT up 7 every time the HR and HL get incremented. You'll see why later. The rest of SUB1Ø is the same, and so is the shifting routine from 122-167.

So what is this JSR $8ØØ? Well, it's a separate vertical shift routine loaded in at $8ØØ, and used along with hor. shifting to create diagonal shifts. But let's look at the differences between TEST K and TEST N before looking at the vertical shifting. TEST N (CALL36934) is the binary file assembled from the source code TEST N --- it does diagonal shifting --- either southwest or northwest. Here is TEST N:

PRINTOUT #39



```
!L     1          ORG $9ØØØ
       2 VT       EPZ $FC
       3 VB       EPZ $FD
       4 HR       EPZ $FE
       5 HL       EPZ $FF
       6 HBASL    EPZ $26
       7 HBASH    EPZ $27
       8 YO       EPZ $6
       9 BASL     EPZ $FA
      1Ø BASH     EPZ $FB
      11 HPOSN    EQU $F411
      12 DRAW     LDY #$9
      13          LDX $7
      14 HERE2    DEX
      15          CPX #$ØØ
      16          BEQ THERE2
      17          INY
      18          JMP HERE2
      19 THERE2   TYA
      2Ø          STA BASH
      21          LDA #$ØØ
      22          STA BASL
      23          LDA VB
      24          STA YO
      25 LOOP11   LDX #$ØØ
      26          LDY #$ØØ
      27          JSR HPOSN
      28          LDY HR
      29          LDX #$ØØ
      3Ø LOOP22   LDA (BASL,X)
      31          EOR (HBASL),Y
      32          STA (HBASL),Y
      33          DEY
      34          CLC
      35          INC BASL
      36          BNE NOCAR2
      37          INC BASH
      38 NOCAR2   CPY #$FF
      39          BEQ NXTLN2
      4Ø          CPY HL
      41          BCS LOOP22
      42 NXTLN2   DEC YO
      43          LDA YO
      44          CMP #$FF
      45          BEQ RET2
      46          CMP VT
      47          BCS LOOP11
```

```
 48 RET2    RTS
 49 STT     JSR $F3E2
 50         LDA #$0
 51         STA $9
 52         STA $C052
 53         JSR $F3D8
 54         LDA #$40
 55         STA $E6
 56         LDA #$0
 57         STA $C054
 58         JSR DRAW
 59         JSR SHIFT
 60         JSR SHIFT
 61         JSR $800
 62         LDA #$0
 63         STA $C055
 64         LDA #$20
 65         STA $E6
 66         JSR DRAW
 67         INC $9
 68         INC $9
 69 LOOP    LDA #$20
 70         STA $E6
 71         LDA #$0
 72         STA $C055
 73         JSR SHIFT
 74         JSR SHIFT
 75         JSR $800
 76         LDX #$2
 77         INC $9
 78         LDA $9
 79         CMP #$8
 80         BEQ SUB10
 81 CONT10  JSR SHIFT
 82         JSR SHIFT
 83         JSR $800
 84         LDA #$40
 85         STA $E6
 86         LDA #$0
 87         STA $C054
 88         JSR SHIFT
 89         JSR SHIFT
 90         JSR $800
 91         LDX #$1
 92         INC $9
 93         LDA $9
 94         CMP #$8
 95         BEQ SUB10
 96 CONT20  JSR SHIFT
 97         JSR SHIFT
 98         JSR $800
 99         LDA HL
100         CMP #$3
101         BCC RTN
102         LDA VB
103         CMP #$BE
104         BCC LOOP
105 RTN     RTS
106 SUB10   DEC HL
107         DEC HL
108         DEC HR
109         DEC HR
```

```
110         LDA #$7
111         STA $CE
112 XYZ     INC VT
113         INC VB
114         DEC $CE
115         BNE XYZ
116         LDA #$1
117         STA $9
118         CPX #$2
119         BEQ CONT10
120         BNE CONT20
121 SHIFT   LDA VB
122         STA YO
123 LOOP33  LDX #$0
124         LDY #$0
125         JSR HPOSN
126         CLC
127         LDY HR
128 START   LDA #$0
129         STA $8
130         STA $CF
131         STA $CE
132         BCC CONT1
133         INC $8
134 CONT1   LDA (HBASL),Y
135         CMP #$80
136         BCC CONT2
137         INC $CF
138 CONT2   LDA $8
139         BEQ CONT3
140         LDA (HBASL),Y
141         ORA #$80
142         JMP CONT4
143 CONT3   LDA (HBASL),Y
144         AND #$7F
145 CONT4   ROR
146         STA (HBASL),Y
147         BCC CONT5
148         INC $CE
149 CONT5   LDA $CF
150         CMP #$1
151         BCC CONT6
152         LDA (HBASL),Y
153         ORA #$80
154         STA (HBASL),Y
155 CONT6   CPY HL
156         BEQ LNDONE
157         DEY
158         CLC
159         LDA $CE
160         CMP #$1
161         JMP START
162 LNDONE  DEC YO
163         LDA YO
164         CMP #$FF
165         BEQ RRTT
166         CMP VT
167         BCS LOOP33
168 RRTT    RTS
169         BRK
170         BRK
171         END
```

The same differences exist between TEST N and K as between M and J. After every doubleshift comes JSR $8ØØ again.

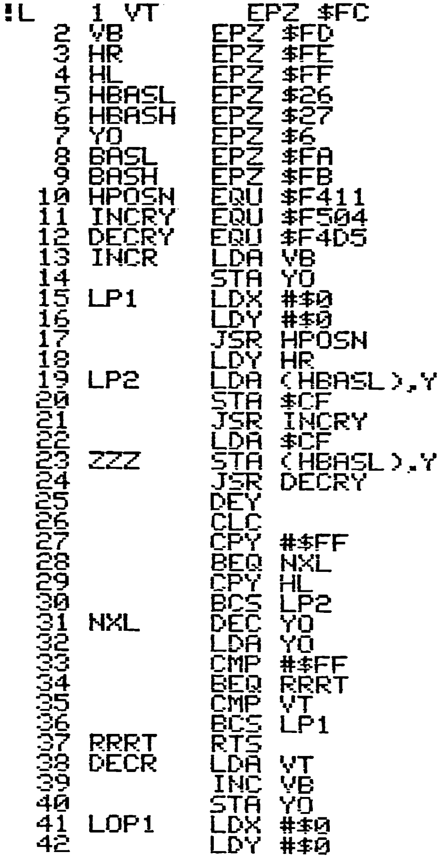
It checks vertical as well as horizontal limits at 99-1Ø5, this time giving a limit of 19Ø.

It increments VT and VB by 7, as we'd expect, if consistency is to happen.

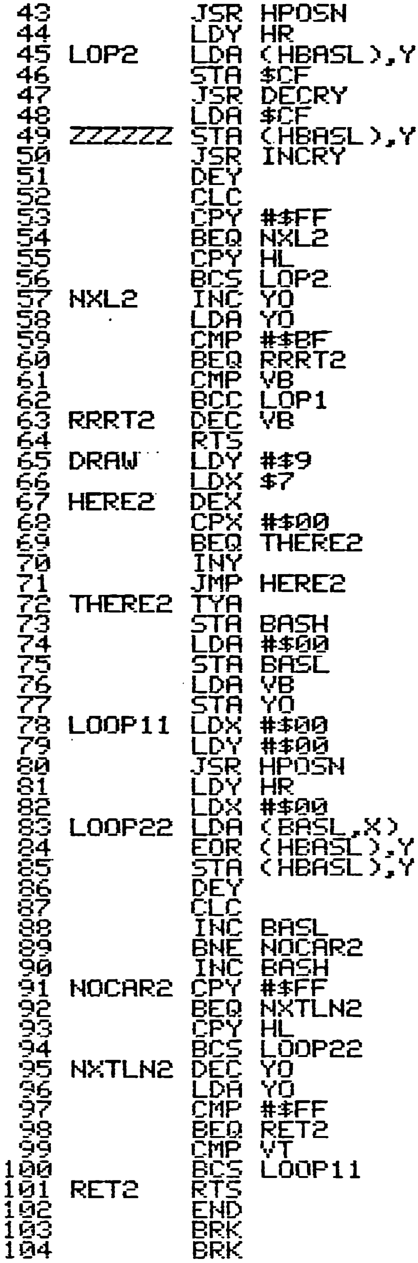
But what, then, is this JSR $8ØØ? It's a routine that was the result of old Don Fudge thinking to himself: if one can shift horizontally, then why not vertically? Of course, there are no 65Ø2 instructions for such a thing, so I had to improvise. (Who ever heard of ROU and ROD instructions?)

I've named the vertical shifting routine TEST L, and the binary file TEST L (CALL2Ø48). So here's TEST L:

PRINTOUT #40



```
!L    1 VT      EPZ $FC
      2 VB      EPZ $FD
      3 HR      EPZ $FE
      4 HL      EPZ $FF
      5 HBASL   EPZ $26
      6 HBASH   EPZ $27
      7 YO      EPZ $6
      8 BASL    EPZ $FA
      9 BASH    EPZ $FB
     1Ø HPOSN   EQU $F411
     11 INCRY   EQU $F5Ø4
     12 DECRY   EQU $F4D5
     13 INCR    LDA VB
     14         STA YO
     15 LP1     LDX #$Ø
     16         LDY #$Ø
     17         JSR HPOSN
     18         LDY HR
     19 LP2     LDA (HBASL),Y
     2Ø         STA $CF
     21         JSR INCRY
     22         LDA $CF
     23 ZZZ     STA (HBASL),Y
     24         JSR DECRY
     25         DEY
     26         CLC
     27         CPY #$FF
     28         BEQ NXL
     29         CPY HL
     3Ø         BCS LP2
     31 NXL     DEC YO
     32         LDA YO
     33         CMP #$FF
     34         BEQ RRRT
     35         CMP VT
     36         BCS LP1
     37 RRRT    RTS
     38 DECR    LDA VT
     39         INC VB
     4Ø         STA YO
     41 LOP1    LDX #$Ø
     42         LDY #$Ø
```



```
        JSR HPOSN
        LDY HR
LOP2    LDA (HBASL),Y
        STA $CF
        JSR DECRY
        LDA $CF
ZZZZZZ  STA (HBASL),Y
        JSR INCRY
        DEY
        CLC
        CPY #$FF
        BEQ NXL2
        CPY HL
        BCS LOP2
NXL2    INC YO
        LDA YO
        CMP #$BF
        BEQ RRRT2
        CMP VB
        BCC LOP1
RRRT2   DEC VB
        RTS
DRAW    LDY #$9
        LDX $7
HERE2   DEX
        CPX #$00
        BEQ THERE2
        INY
        JMP HERE2
THERE2  TYA
        STA BASH
        LDA #$00
        STA BASL
        LDA VB
        STA YO
LOOP11  LDX #$00
        LDY #$00
        JSR HPOSN
        LDY HR
        LDX #$00
LOOP22  LDA (BASL,X)
        EOR (HBASL),Y
        STA (HBASL),Y
        DEY
        CLC
        INC BASL
        BNE NOCAR2
        INC BASH
NOCAR2  CPY #$FF
        BEQ NXTLN2
        CPY HL
        BCS LOOP22
NXTLN2  DEC YO
        LDA YO
        CMP #$FF
        BEQ RET2
        CMP VT
        BCS LOOP11
RET2    RTS
        END
        BRK
        BRK
```

Notice that in line 11-12 I define INCRY as $F504 and DECRY as $F405. These aren't Fudgisms. They're Appleisms. These are bonafide Applesoft subroutine entry points. JSRing to the INCRY routine moves your Y coord. up 1 and DECRY moves your Y coord. down 1, and either appropriately change $26 + $27 base addr.

Line 13-17 starts out the same as the shifting subroutines. Store the VB in $6 and do the HPOSN routine at $F411 to get your proper hi-res address, before indexing, into HBASL ($26) and HBASH ($27).

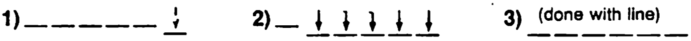
Line 18 puts HR into Y, as this will be our address indexer.

Line 19-20 loads proper hi-res byte into the accumulator, and saves it in $CF.

Line 21 the Y coord. is incremented and the hi-res address is now quite different, in $26 and $27.

Line 22-23 loads accumulator with the hi-res byte saved in $CF. The INCRY routine disturbed accumulator contents. This byte is loaded into the incremented hi-res address.

Line 24 puts the address in $26 and $27 back the way it was before it was incremented by one Y coord. This is so we'll be at the correct line of bytes (the VB line, in this case) all the way from HR to HL. Think of moving down the bytes like this:



What we're doing here is moving the block-shape's bytes downwards, one at a time, until a line is done. Then we move the next line down to where VB used to be, and we keep going, until the VT line ends up at VT + 1 and the entire shape is now "shifted" one dot downwards.

Line 25 move left to the next byte on the VB line. This will be the next hi-res byte we move down to the line underneath it.

Line 26 we're making sure the carry isn't wrong.

Line 27 see if the last hi-res byte moved down was byte #0, which would mean that DEY in line 25 has now decremented that 0 down to $FF --- yes, folks, that **is** what happens when 0 is decremented --- one can interpret 11111111 (binary) as 255 or -1, the former being the simple addition of bits according to their place value, the latter being the complement version of minus one. See an assembly book.

Line 28 if the last byte was #0 we branch to the next higher line (the next lower Y coord.).

Line 29-30 checks to see if you're at HL yet; if you are, go on to NXL; if not, you loop back to LP2 (line 19) and get the next byte leftward (towards HL) and move **it** downwards.

Line 31-32 here we decrement the temporary Y coord. value to correspond with the next higher line, and then load this # into the accumulator for 2 reasons: the checking in line 33-34 and the HPOSN routine we'll be looping back to soon. If you'll recall, HPOSN requires Y coord. in accumulator upon entry.

Line 33-34 checks to see if the last vert. coord. was 0; if it was you're done so exit routine. (see comment for line 27).

Line 35-37 checks to see if we've just performed our move-down on the block-shape's top line; if we have, we exit routine --- otherwise we move up to the next line.

Line 38-64 will "shift" a block-shape down, rather than up:

Line 38-40 loads permanent VT into temporary VT ($6), and increments VB so that lines 61-62 work right. You see, BCC will not branch you unless the accumulator's **less** than the data. What was needed here was an instruction that branched you if the accumulator was less than **or equal to** the data. I improvised on the VB variable's value so that the BCC **would** act like equal to or less than.

Line 41-58 are the same as lines 15-32, except that the DECRY and INCRY routines are interchanged, and YO ($6) is incremented, not decremented, since as we move upwards we'll be going from higher block-shape lines to lower block-shape lines in our byte-moving operations.

Line 59-60 checks to see if you've reached the bottom of the screen (191); if you have, then the routine is exited.

Line 61-64 checks to see if we're at the bottom line of the shape yet; if we're not, we're sent back to LOP1 (line 41) to move down a line and begin moving bytes upwards again.

13C. COLOR AND DOUBLE-SHIFTING

TEST H and TEST I were merely shifting-routine programs --- they could have their shifting (technically, the bytes are **rotated** but the shapes get shifted) routines used in either one or two-page animation.

TEST J and TEST K are two-page double-shifting color-safe routines. Shapes move 2 bits sideways, therefore color is consistent. If you wanted 1-bit shifting merely substitute NOP for each second JSR SHIFT line in the source code, and EA EA EA in place of the 20 (lo. byte) (hi. byte) in the binary file (there are other lines that would need changing too --- see below). The 20 is the **opcode** meaning JSR and the low and high bytes are called **operands.**

The easiest way to change a routine like TEST N (CALL36934) is to EXEC it into LISA and delete the extra JSR SHIFT lines and the extra DEC HL and DEC HR commands in lines 107 and 109. Then hit A to ASSEMBLE the file and then re-WRITE the source-code file (TEST N) and resave the binary file by typing:

```
CTRL D BSAVE TEST N (CALL36934),A36864,L333
```

Remember that you'll have to stick to white shapes and you may need to do a few other alterations of the routine, but it will get smoother.

A good project would be to make any of these shift-animation programs do XDRAW --- any takers?

TEST M and TEST N are 2-page double shifting color-safe routines that do triple-shifting: two horizontal shifts and one vertical "shift." There's so much going on they're rather slow routines unless you use fairly small shapes. These routines use TEST L (CALL2048) for their vertical shifts. It loads at $800 while TEST M (CALL36934) or TEST N (CALL36934) load at $9000 and need HIMEM:36864.

It's probably obvious that TEST L is not really a shift-program in the strict sense, since it uses no logical shift or rotation. But it **is** a shape shifter --- it shifts shapes up or down according to needs, and by a byte-at-a-time method that's very similar to the horizontal shifters.

The INCRY and DECRY routines of Applesoft have many other uses in assembly graphics, TEST L is a sample of how to deal with it. One could use YTABLE for the incrementing and decrementing of Y coords., rather than HPOSN or INCRY or DECRY. Sometimes these latter two routines are tricky to use in conjunction with animation routines and vector, block or hplot-shapes, since only the "internal cursor" is affected, which isn't always enough. See later in book for more YTABLE information.

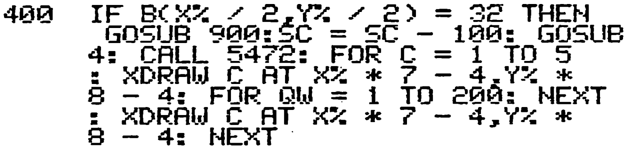
14 ANIMATION FROM APPLESOFT/ ASSEMBLY/COMBINATION

14A. APPLESOFT

An all-Applesoft animation would be exemplified by the way explosions happen in the SAMPLE GAME W/VIOLIN & NOISES program on 28C. BOOM is the vector shape table that gets its shapes drawn and erased in a way that looks like an explosion. The actual explosion sounds are machine language, but the visual effects are all Applesoft. Sometimes Applesoft suffices --- other times it doesn't.

What is all-Applesoft animation like? Here's how it happens in the above explosion:

PRINTOUT #41



```
4ØØ  IF B(X% / 2,Y% / 2) = 32 THEN
      GOSUB 9ØØ:SC = SC - 1ØØ: GOSUB
     4: CALL 5472: FOR C = 1 TO 5
     : XDRAW C AT X% * 7 - 4,Y% *
     8 - 4: FOR QW = 1 TO 2ØØ: NEXT
     : XDRAW C AT X% * 7 - 4,Y% *
     8 - 4: NEXT
```

The 2-dimensional array B (X%/2,Y%/2) is the one representing the "playing board" screen's squares. A "32" means a "mine" (invisible) is on that square. GOSUB 900 is where I check to see if there have been 5 mines landed on since the last "chance tornado" (see instructions). If there have, you get another one.

CALL5472 runs the explosion sound. EXPLOSION (CALL5472) is loaded in at 5472. The GOSUB 4 previous to this is the first **part** of the explosion sound, generated by a mach. lang. noise routine. Shapes (vector) 1-5 get DRAWN and XDRAWN, with 1/4 second delay loops.

You see why it's easy to animate in Applesoft? (Not so easy, though, is meeting **speed** requirements.) Simply draw and erase into BASIC commands.

For more information on pure Applesoft animation, you may see the SUPER SHAPE DRAW AND ANIMATE package we sell, or list out 6 of 28A and pretend each mach. lang. block-shape-drawing CALL is a vector Applesoft XDRAW command. Use E or F of 28B for doing listings.

148. ASSEMBLY

Most of the animation in this system is mach. lang., although to simplify matters I've used BASIC programs to get the animation going. Some programs, like 5 of 28B, use a combination of BASIC and Assembly --- the BASIC may set up a FOR-NEXT while the mach. lang. routines merely do the drawing and/or shift-movements.

In most cases you could convert the BASIC parts to Assembly and/or the Assembly parts to BASIC. You can also load a mach. lang. program with memory without ever BLOADing or even creating a mach. lang. file. All you need to do is POKE it into memory. The easiest way to do this, if you have many hex codes to get into memory is to convert your mach. lang. binary file to a list of decimal numbers that you put into a data statement. READ in the data and POKE it into memory. If you want to put whatever is in $6 into the Y register, and this will be done in $320, then you need an LDY $06, which is, in hex codes, $A4 and $06. The decimal #s here are 164 and 6. Since $300 is dec 768 and $20 is dec 32, then $320 is dec. 800. So POKE 800,164:POKE 801,6 will give you what you want. Or:

```
100 FOR X=800 to 801:READ Y:POKE X, Y: NEXT
110 DATA 164,6
```

In $322 (802 dec) could be a $60 (96 dec), which is RTS. All you need to do now is CALL 800. (Change 801 in line 100 to 802 and add 96 to line 110.

Isn't BASIC neat? So much can be done so easily! You just fantasize something you want to happen and --- Presto! --- you can program it in a few minutes (or hours).

Of course, the problem here is **speed**. When you fantasize things moving real fast and then you try to get BASIC to help you create these fantasy worlds, you're in a pickle. BASIC is great for when speed is NOT important (most of the dozens of games I've written are not speed-centered and didn't need mach. lang. very much, except for sounds --- most good sounds or hi-res scrolling require mach. lang.).

So most of the mach. lang. routines on these disks involve using BASIC programs, but only to input "what shape table?", "how many steps per shape?", "what are the starting coords.?", "what are the limits?". After that the routines are called and you're in pure mach. lang. In actual programs that you create, you needn't have any BASIC driver programs. Simply put all these values into your binary file to begin with, and if shape tables need to be loaded in, either do it from mach. lang. or make that part of the booting program's job.

If a source code starts out 1 VT EPZ $FC then that means that your binary file needs to always store it's block-shape's vertical top coordinate in $FC, and when it gets its initial value for VT it should load it into $FC:

```
LDA  #$0A
STA  $FC
```

The above instructions would stick in a 10. If you **did** use a BASIC driver, POKE 252,10 would do the trick.

You may notice that some of my BASIC drivers POKE a considerable number of numbers into memory before they begin. Two things are happening here:

1) Addresses used in the binary file to hold things like "current shape #" are loaded up with the proper initial shape #, which may or may not be altered later. Example: POKE 7,2.

2) Contents of addresses right in the binary file itself containing either opcodes or operands (both of these are **object codes**) are changed if you've decided to go left in a rightward animating program, for example. Perhaps DEC($C6) would be substituted for INC($E6) and BCC would replace BCS and SEC would replace CLC and SBC would replace ADC. This is simple to do and makes 1-function programs turn into 2-4 function programs. Why make 4 mach. lang. programs if adjusting one program will do as well? (However, sometimes you really do need separate complete routines for similar but nonidentical functions.)

(Sometimes what logically seems like a good candidate program for changing into a 2-function program via POKES or extra LDA and STA commands turns out not to be at all appropriate for the change. For instance, TEST H and TEST I were created to shift shapes right and left, respectively. There was an obvious tendency to want to change a few commands, like ROL to ROR, to change TEST H's shift direction, and then not bother to create a separate left-shift program. But it turns out that the routine it takes for left-shift is quite different than the right shift routine!)

More on Assembly later in this manual.

14C. COMBINATION

Why use both Assembly and BASIC in a program? Well, suppose you have a fast-moving space war game that has several paragraphs of involved instructions? The really fast action will all have to be in mach. lang., but try·writing a lot of instructions in mach. lang. --- a useless hassle. Do them in BASIC.

A complicated strategy game with hi-res shapes being moved fairly slowly is far easier to do with BASIC --- if you can get a compiler like the one ON-LINE sells to turn BASIC into Assembly in a way that's suitable for decent game-making, that's fine, but to actually **write** it in Assembly in the first place --- a needless hassle.

(Compilers will not turn slow BASIC hi-res games into fast arcade games. There will be a speed improvement, but it won't be that large. Compiled programs must be shorter than BASIC ones and they are very memory-inefficient. They'll never replace assembly for arcade speeds.)

One reason for a "combination" program is to render existing programs multi-functional. (See preceding section.) It's simple to change object code via POKES, and it's simple to do it with LDA and STA instructions (you don't even have to convert hex. to dec.), but sometimes if you're going to jump around between various mach. lang. programs, it's easiest to CALL them from BASIC.

Often the only mach. lang. routines a program needs are in the area of scrolling (hi-res) or sounds. (**Superfont** is my example of the former situation, and **Sample Game W/Violin & Noises** is my example of the latter situation.)

15 DRAW vs. XDRAW

The way one draws is to put something somewhere. The way one XDRAWS is to put something somewhere IF the bits you draw on are not like the ones you're trying to put there. XDRAW puts the **complement** (opposite, such as Black vs. White or Green vs. Blue) of the color already existing at each point plotted. If you're using a white background and you XDRAW a shape, it'll be black the first time, but the second time you draw it it'll be white, therefore erased and invisible. The same would happen with blue on a green background or white on a black background. The plan is:

1) the 1st XDRAW draws the shape
2) the 2nd XDRAW erases the shape

In BASIC, XDRAW is like this (use 28C here):

```
HIMEM:36864:POKE 232,Ø:POKE 233,144:
?CHR$(4);"BLOAD SPACESHIPS":HGR:
POKE 163Ø2,Ø:SCALE=1:ROT=64:
XDRAW7AT99,99:FOR QW=1TO2ØØØ:NEXT:XDRAW7.
```

You probably didn't need the extra commands above, but in case you needed a refresher, there they are. The HIMEM keeps the shape table out of harm's way. The 2 POKES say "let's have the computer realize that my shape table starts at $9ØØØ(36864), which keeps it away from the main program, which starts at $4ØØØ due to POKES you'll have done in Hello: POKE1Ø3,1:POKE1Ø4,64:POKE16384,Ø. (If you started at $8ØØ, the normal program-starting place, you'd have only 24 pages of memory available. Even if you used HGR2 and laid off HGR (page 1) you'd have only 56 pages available (5Ø sectors on disk). But if you start at $4ØØØ and use HGR for your hi-res graphics, you'll have from $4ØØØ-$96ØØ or 86 pages (78 sectors on disk), unless your HIMEM is at $9ØØØ, which will leave you with 8Ø pages or 72 sectors.)

The POKE 163Ø2,Ø gives you full-screen page one graphics. The first XDRAW draws, the second XDRAW erases, after a delay loop of 2 1/2 seconds. The second XDRAW needs no coordinates since it will use the ones already stored (X=99,Y=99). It's often a good idea to play cautious and use coordinates even when they seem optional ---you'll usually find some reason why you'll have changed stored coordinates.

Let's examine mach. lang. "drawing" of a block-shape, which is merely a transferring of bytes from a table to hi-res address:

```
LDA  (BASL,X)
STA  (HBASL),Y
```

The first command loads the accumulator with the data found in the address pointed to by (BASL,X). BASL is the address where the low byte (followed by the high byte) of the shape table address is stored.

byte (followed by the high byte) of the shape table address is stored. This is called **indirect** addressing because we're going to an address to find the address of data rather than the data itself and it's called **pre-indexed** indirect addressing because the indexing is performed before the indirection. In other words, the X register is added to BASL to get the final address of the table byte needed. As it happens, we could increment either BASL or X to get other table bytes later. In these programs the former was normally chosen, because the X register was often changed before re-using the LDA line, while BASL was untouched, therefore convenient. So X was kept at 0 for this instruction.

The second command stores the table byte in the appropriate hi-res byte. It's again an **indirect** situation because it's an address containing an address, not data. But this time it's **post-indexed** indirect addressing because the indexing is performed after the indirection. In other words, we go to the HBASL address to find out the address of the proper hi-res line, but then to get the proper **byte** as well, we must move right until we're at the correct byte. This is a forward displacement accomplished by indexing with the Y register.

All these commands do is take from a table and give to the screen, for viewing. The **way** they do it is to ignore what's already at the destination address. They'll jump right "on top" of other shapes already occupying the screen bytes, wholly or partly. If this is the type of drawing you want, then in source codes change all lines to NOP (no operation) that now say EOR (HBASL),Y, or in binary files, put EA EA in place of all sets of numbers that look like this: 51 26.

What if we change the "drawing" commands above so that EOR (HBASL),Y is included? We'll get XDRAW rather than DRAW, in effect.

With EOR, logical either-or, we get ON bits only when 2 differing bits are compared. If we "draw" a 0 on a 1 or a 1 on a 0, we get a 1 (ON), but if we draw a 0 on a 0 or a 1 on a 1 (erasing), then we get a 0. Naturally this is also how the Applesoft XDRAW command does it's thing.

In XDRAW ($F65D) there's both $F695 JSR $F49C and $F6A1 JSR $F49D. Either of these will send you through $F4C4 EOR ($26),Y and $F4C6 STA ($26),Y. And perhaps even through $F4C2 INC $EA --- see collision counter section (next).

Now try running 3 of 28A and loading in **vector** shape MAN and viewing shape #5 (the flying saucer) in both DRAW and XDRAW mode. See the gap in XDRAW mode? That's because this section got plot vectors twice, when it was created. So in XDRAW it turns on that section but then turns it off again since we're EORing with 2 ON (1) bits.

There's one more important point to consider about XDRAW ---there **is** a popular alternative to XDRAW which is used in many arcade games.

Suppose you're in a Space Invaders type of game and those cute little aliens are moving across the screen in rows. The following method would preclude the need for XDRAW:

DRAW a solid green shape of a monster. Then DRAW it again only this time move it over 2 dots. The monster/alien will now be wider. Suppose you're moving this "invader" left. Your monster will look now as though s/he has gotten wider to the left, but his/her right side will remain unchanged. But there's a way to deal with this problem.

Notice precisely what part of the **old** shape is still visible --- merely a thin green sliver that's like a backwards "C." (Perhaps there are a few "legs" areas that have changed also --- this would make this alternative animation method less viable but still possible).

What is the background color? Let's say black. If you were to draw that C-sliver in black, then your shape would have "moved," for all intents and purposes.

So animation, using vector shapes, would be a matter of needing no erasing or XDRAWing, but merely drawing an alien in green, and then another in a shifted position, and then drawing the black C-sliver, and then the alien --- shifted over even more, and then the black C-sliver.

The benefits of not XDRAWing, in this situation, are quite obvious: DRAWing a little sliver of a black shape is a faster proposition than erasing and re-DRAWing an entire alien shape. Speed-wise you're much further ahead to do the black C-sliver. Do it immediately after the second shape is DRAWN over the first one.

Incidentally, you can do the same thing even faster by using block shapes. Use 2-bit moving with a 7-shape sequence which has all its components 2 bits apart, for a total of 14 bits of movement. Use 2-bit moving rather than one-bit moving to preserve color integrity (see Chapters 13 and 17). Use shape #s 1-7 for aliens and #s 8-14 for black C-slivers. Make shape 9 fix the alien just after 2 is superimposed upon 2, etc. All EOR (HBASL),Y instructions will need to be NOPed with EA EA (no operation) in the block-shape drawing routines (or use POKE addr.,234) so that the block shapes DRAW, not XDRAW.

Black C-slivers will be 1 byte wide and several lines high, and will include part of the green alien.

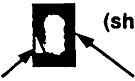
1)

(shape 1)(green)

2)

(shape 2 onto shape 1)(green)

3)



(shape 9 lops off visible part of shape 1)

Block-shapes are already so fast, however, that the sliver/DRAW method is probably of little value.

THE COLLISION COUNTER (SEE CHAPTER 8G)

16

The collision counter is the address $EA, which is 234 dec. Run E of 28A. List the program out, also, so you're sure you know what's making the changes in the collision counter. A change means a collision. This is convenient because all one needs to do is store a value (CC=PEEK(234)) after drawing a shape and then monitor that address at appropriate intervals to see if other shapes in the vicinity have hit (changed) it:

```
900 if PEEK(234) < CC THEN GOSUB 29000.
```

The uses for such a collision detector are limited only by your own imagination. If the collision represents colliding with an explosive mine shape, then you can CALL EXPLOSION:GOSUB (to shape blow-up sequence:SCORE=SCORE-1000:COLLISION=COLLISION +1:etc. EXPLOSION would be a machine language sound routine, and all of the set of commands above would be at GOSUB 29000. In COLLISION (a variable that's only actually CO in Applesoft --- the other letters don't count) you'll have the # of collisions until now. In SCORE you'll have the consequences (-1000) of your mishap.

There are addresses where you can see $EA being dealt with: $F622 in DRAW and $F67E in XDRAW and $F4C2 or $F4AB, in the end of INCRX, which HPOSN sometimes branches to because of the BPL $F48A in $F465.

Examination of the XDRAW routine, with the branch to $F49C or $F49D that lets you do INC $EA according to how the shape gets drawn, gives you some hints that INC $EA would happen a different # of times if XDRAW were happening partially on top of another shape. But these technical details are of no consequence for the immediate purpose: the understanding of the way to use PEEK(234) as collision monitor.

17 HI-RES COLORS - THE PALETTE PROGRAM

Six colors are normally available on the hi-res screen: HCOLOR 0-7, with 0 and 4 both being black and 3 and 7 both being white. 1 is green, 2 is violet, 5 is orange, 6 is blue.

The color masking table byte for black is 0's, so it's easy to see why this gives a black color, and why white, which has ones for its color masking byte, gives white. It's a bit harder to see why the 4 colors, which are alternating ones and zeroes in the color masking byte, end up being produced.

Let's look at HPLOT now. What happens in HPLOT is first a JSR $F411, which is the HPOSN routine which finds the hi-res address to correspond to your coordinates (and then gets $1C ready with the color bytes and $30 ready with the bit position). Next LDA $1C gets the color byte into the accumulator so that it will properly mask the point to be plotted. Next EOR ($26),Y masks the entire byte at the hi-res address/screen position which includes the bit/dot to be plotted --- the results end up in the accumulator. Next AND $30 allows only 1 bit to get turned on in the plotting. In the AND command, if either operand is a 0 the result is 0. And all the bits are 0 in the bit position byte that's in $30 except bit 7 and the plotting position bit. (Bit 7 is 1 because then ANDing at that position assures that the color bit (#7) will be let alone, whether it's 1 or 0.) Now the accumulator contains a byte which, when stored on the hi-res screen, will show only one visible dot, if any (depending on whether the bit position's byte's ON bit got ANDed with a 1 or a 0 in the masking byte that was struck in the accumulator.

Now we EOR ($26),Y **again**! Why? This is to put the plotted dot in with the hi-res byte that was already there before we ever started HPLOT. Since the accumulator contains only 1 (or 0) "dots" in a visible position now, EORing with the old, original, hi-res byte will have the effect of **producing a byte that is the original hi-res byte plus the results of the plotting.** All that remains to be done is STA ($26),Y which stores the amended original byte in its proper hi-res position.

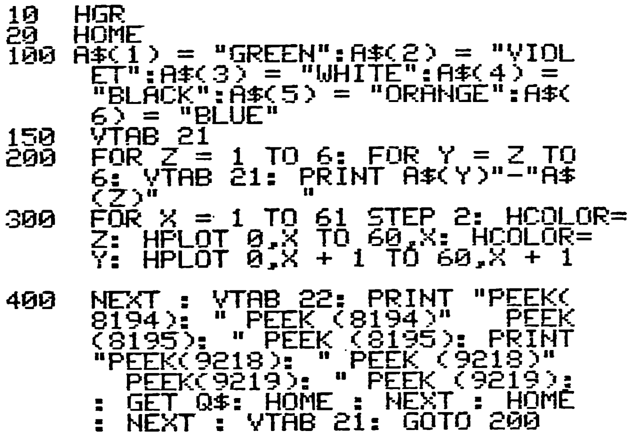
Remember, EOR gives a one only if the bits compared are **different.** This means an old byte with 3 bits ON in positions that don't correspond to the plotting position dot would turn into a new byte with those same 3 bits on **plus the newly plotted one too.**

Let's see the routine again:

```
HPLOT:  $F457   JSR  $F411
                LDA  $1C                       color byte
                EOR  ($26),Y                         mask
                AND  $30                     bit position
                EOR  ($26),Y  add plot to original byte
                STA  ($26),Y                 display plot
```

Now let's test a program:

PRINTOUT #42



```
10  HGR
20  HOME
100 A$(1) = "GREEN":A$(2) = "VIOL
     ET":A$(3) = "WHITE":A$(4) =
     "BLACK":A$(5) = "ORANGE":A$(
     6) = "BLUE"
150  VTAB 21
200  FOR Z = 1 TO 6: FOR Y = Z TO
     6: VTAB 21: PRINT A$(Y)"-"A$
     (Z)"        "
300  FOR X = 1 TO 61 STEP 2: HCOLOR=
     Z: HPLOT 0,X TO 60,X: HCOLOR=
     Y: HPLOT 0,X + 1 TO 60,X + 1

400  NEXT : VTAB 22: PRINT "PEEK(
     8194): " PEEK (8194)"   PEEK
     (8195): " PEEK (8195): PRINT
     "PEEK(9218): " PEEK (9218)"
       PEEK(9219): " PEEK (9219):
     : GET Q$: HOME : NEXT : HOME
     : NEXT : VTAB 21: GOTO 200
```

This program is 7 of 28D.

Perhaps you're wondering what those PEEKS are for. They show 4 bytes --- in the following places (dec.):

9218	9219
1Ø242	1Ø243

All 4 bytes are adjacent --- notice that moving down a line raises the address by 1Ø24, as it does 7/8 of the time. Also notice that horizontal addresses are consecutive.

The reason for looking at the screen bytes in these 4 addresses is simple --- we want to monitor what types of colors are created on color monitors when various bytes are used. Why not just one byte? Because it takes looking into **at least** 4 bytes, as above, to see what's going on with colors. But don't take my word for it --- run the program until you've seen lots of different combinations. Make sure you tune in your color monitor so it matches the colors you're supposed to be seeing --- turn the tint button so green is green, not orange --- etc.

What do you make of the fact that the colors are composed of different bytes? Why is orange-green a color that has 4 different bytes? Simple enough: the odd color bytes (#1,#3,#5, etc.) get EORed with 127 (dec.) which is Ø111 1111 in binary and $7F in hex. The result of this EORing is getting the **complement** of the desired color, or it's as if the byte ASLed or LSRed --- shifted sideways so that all the zeros and ones exchanged places. Confused? Don't worry, I'll hang in there with you until it seems clear.

Run program #1 on 28D until violet-violet is displayed. If you have no color monitor don't sweat it --- you'll still be able to learn about colors.

Violet-violet has 85 and 42 for its bytes. But the color mask for violet formed in $F6F8 of the color masking table is only 85 --- there's no 42 about it.

green:	$F6F7: HCOLOR=1	$2A	00101010	42 dec.(32+8+2)
violet:	$F6F8: HCOLOR=2	$55	01010101	85 dec.(64+16+4+1)

The 42 and 85 are **complementary** binary bytes. If you think of the hi-res screen as 280 columns that are 192 dots tall, things get easier. On a color monitor if a dot is in an odd column, it will appear green. If a dot's in an even column, it will appear violet. Can you see why 01010101 is the violet masking byte now? If we started in byte 0, at the left edge of the screen, and loaded in that byte, it'd look violet. The ON bytes would be at hi-res hor. coords. of 0, 2, 4, and 6 --- all even, and since even column dots are violet, we've just made a violet byte-line (a byte-long-line).

Okay, if you're screaming at the top of your lungs that "Fudge is nuts! Those coords. would be 1, 3, 5, 7." I'll remind you of what you forgot:

1) bytes on the screen have an invisible color bit --- bit 7.

2) the screen bytes are all displayed backwards.

So the above 01010101 would show up as 1010101 on the hi-res screen, just as old Fudge-face told you. (I'd better quit talking about myself that way --- the insult I don't mind, but it makes me **hungry!**)

(Incidentally, the way EOR #$7F gets a complement is that it compares the 7 visible bits with all ones and only when bits differ will the result be a one. So when the accumulator has a 1 bit the result is a 0 and when the accumulator has a 0 bit the result is a 1. So the opposite, or complement, is the result here.)

Now 9218 and 10242 are even numbered bytes that correspond to byte #2 on the screen, so 1010101 really would be a violet line because its ON bits would be at 0, 2, 4, 6. This is probably clear now. If it isn't don't go on. Reread this chapter until the fog lifts. Your Apple Reference Manual (white) has some of this information on page 19. Check it out.

But what has not been clarified is this: why must the bytes in odd byte #s be shifted or complemented with the instruction EOR #$7F? Why can't "85" be violet at odd bytes too? Well, let's put 1010101 in byte 1 position. The ON bytes have coordinates of 7, 9, 11, 13. This would be **green,** not violet, because the ON bytes are in odd columns, not even ones. Don't forget, now, that the visible 7 bits in byte 0 were potentially #0-6, which is a total of 7. The next available visible bit is then #0 bit of byte 1, which will have a hor. screen coord. of 7. So 1010101 really would be ON bits at odd columns.

So the **complementing** done at all odd byte (#1-39) addresses is to **cure** the above problem. All it would take to make 1010101 go from green to violet is to shift the bits over 1 or, easier to deal with, get the byte's opposite (complement, where ones replace zeros and zeros replace ones). By reversing the "85" masking byte after removing its 7 bit we see what its screen byte would be like (on-screen 1010101), and by complementing it we can see that on-screen the complement of the 85 byte, **needed at each of the 20 off-byte positions on the screen,** would be 0101010. And if 0101010 were found at the 1 byte (hor. coord.) then we'd have the ON bits at 8, 10, and 12 --- and guess what color that'd make these dots? Yes --- violet! No matter what, these dots at even columns on the color monitor (if color bit is 0) will be violet.

So the bytes on each line will be 85 42 85 42 85 etc., when a violet line is displayed. Change the bytes as the 85 and 42 bytes are exchanged and you get a green line --- the complement. I'm talking about **horizontal** lines here.

Now, everything I've said above applies to the color bit (#7) being off in all bytes. If it happens to be ON in all bytes then you can replace violet and green with blue and orange, respectively. Blue is 213($D5) and orange is 170($AA). Bytes of 213 170 213 170 will be blue lines, if the first byte is even, and orange, if the first byte is odd. Notice the ON color bits:

orange: $F6FB: HCOLOR=5	$AA	10101010	170(128+32+8+2)
blue: $F6FC: HCOLOR=6	$D5	11010101	213(128+64+16+4+1)

Now, when 2 dots are side by side, the result is white. Notice that white masking bytes are all ones and black masking bytes are all zeros (not including the highest bit's setting).

black1: $F6F6: HCOLOR=0	$00	00000000	0 (0)
white1: $F6F9: HCOLOR=3	$7F	01111111	127 (64+32+16+8+4+2+1)
black2: $F6FA: HCOLOR=4	$80	10000000	128 (128)
white2: $F6FD: HCOLOR=7	$FF	11111111	255 (128+64+32+16+8+4+2+1)

Perhaps it's now obvious that in plotting or drawing things on the hi-res screen, there'd better be some way of distinguishing which byte addresses are odd and which ones are even --- and the odd ones will need their color masking bytes complemented with EOR #$7F because of the fact that:

1) 2 colored dots side by side always appear white
2) dots in even columns are black, violet, or blue.
3) dots in odd columns are black, green, or orange.

Let's look at how the odd-byte problem is handled:

$F44F LSR the hor. byte coord. has been stuck in here and now we'll see if it's odd or even. We shift the bits right, and the 0 bit, which has a value of 1 if ON, and is only ON in odd numbered bytes, gets shifted into the carry register (C).

$F454 BCS $F47E if the carry was set, the byte address was odd so we branch to $F47E to do the complementing.

$F47E ASL here we're shifting the color masking byte left so 2 consecutive bits, #5 and #6, can be evaluated. What we're looking for is 2 ones in a row in the visible (0-6) part of the masking byte, which will mean the color **white,** which will cause us to exit with RTS.

$F47F CMP #$C0 if the 6 and 7 bit (the 5 and 6 bit before the ASL) are both one, then the value here is at least 192 or $C0, so we see if white is the color by comparing the shifted color masking byte with 192. C,P subtracts the data from the accumulator but doesn't store the result --- it merely sets flags N, Z and C. It's the N flag we'll be checking here.

$F481 BPL $F489 if 192 subtracted from the shifted accumulator byte would have a plus result, or a zero result, then here you'll be sent to RTS. N flag=0 means result= 0. BPL branches if N (negativity)=0.

$F483 LDA $1C load in color masking byte if the color wasn't white; prepare for the complementing.

$F485 EOR #$7F complement the color masking byte.

$F487 STA $1C stick the complement in the color masking byte's address. $1C is the "internal" cursor's storing place --- the "external" cursor's place is $E4.

$F489 RTS exit

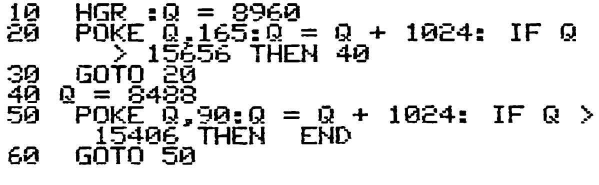
It should now make sense that the values of orange-green are:

42	85
170	213

Green's mask byte is 42; 42 has a complement of 85. Orange's mask byte is 170; 170 has a complement of 213. Since we're starting each of the above at even hor. bytes, we needn't reverse the above #s with their complements. If we **were** to do this reversing, our color will have changed to blue-violet.

Page 19 of the Reference Manual (white) tells us we may as well forget mixing non-complementary colors. Let's see if that's true. First off, a mixed-color (green + orange or violet + blue) byte would involve the high bit in both the on and off positions at the same time --- rather difficult! Well, that kills that. But how would complementary colors be mixed in a byte? Easy: don't alternate 0 and 1 evenly. Try 10100101 or 01011010, which in decimal would be 165 and 90. The first gives orange and blue stripes and the second gives green and violet ones, in the following test program:

PRINTOUT #43



```
10  HGR :Q = 8960
20  POKE Q,165:Q = Q + 1024: IF Q
      > 15656 THEN 40
30  GOTO 20
40 Q = 8488
50  POKE Q,90:Q = Q + 1024: IF Q >
     15406 THEN  END
60  GOTO 50
```

Then change line 4Ø to Q=8961. See? --- you can invent color patterns all day.

Now let's go to the **Palette Program,** 4 on 28D. If you have anywhere near as much fun drawing and painting and filling the screen with colors and inventing new colors as I did, then get set for a really good time. Not only was FILL 1 a really exciting assembly challenge and fun machine language algorithm to create, but PALETTE was a real ball as far as BASIC programming goes. Near the beginning of the program, from 8-8Ø, is a color permutation subroutine. Then around line 3ØØ is a simple 21-color routine that's sort of a standard for many color programs. Run option 1Ø in Palette (4 of 28D) and see what it's all about. It takes more than a minute to draw on hi-res page 2, but from that time on you merely flip pages and you'll see the color palette instantly.

Yes, Virginia, you get to "fill" with all 181 of those colors! But you also get to fill with mystery colors. Run option 17.

What do I mean "fill", and what's a mystery color, right? All right ---let's go through it, as they say in basic training, by the numbers.

1) To fill means to indicate a starting point within the boundaries of a part of a picture, and then begin coloring it in with chosen colors or patterns of colors.

2) Use option 9 now and then choose option 3 of 9 which will allow you to load in a 34-sector picture from disk. Ask for COMPOSITE 2 which is from our **Chambers of Xenobia** adventure (it was drawn with both Instant Graphics (Block Shapes) and Super Shape Draw ---the former is on 28A and could have done the whole picture easily ---the latter is available from us).

3) Use option 12 to get full-screen graphics and suit yourself about option 18, which will double vertical line widths so they look whiter and better. (Incidentally, when you're in option 9, choice 1 or 2, 8 of 28D, you'll use button #1 to make shapes appear and button (PDL)#Ø to exit. In 4 of 28D use button #Ø to draw.)

4) Now look at the picture with option 1. Then choose a random mystery color with option 17, and try option 16 to remind you of your current color bytes. You want to see what your mystery color looks like? Go for option 11, the filling subroutine. If you don't like the clicks, hit A to abort clicks.

5) Notice that there are lots of bounded segments of the picture and you can't get from one segment to another without crossing lines. The fill routine is set up to fill most or all of each segment you enter with the paddle-directed dot-cursor and hit PDL button #Ø. If the top and bottom sides of the segment are horizontal and neither of the left or right sides of the segment interfere with upward or downward travel (from the start-point), then the entire segment will be filled almost instantly with your latest palette or mystery color.

6) Notice how the filling works (there's a whole chapter on it later): The color fills in upwards until a non-Ø byte (with no room for a 2-bit color pixel)*is encountered (in a straight-up direction from the starting point). Then the unfilled portion of the segment **underneath**

*A pixel is the smallest visible color dot, the smallest unit of resolution currently dealt with; i.e. 2 consecutive bits for hi-res color which is 14Ø resolution.

the dot-cursor starting point gets filled until **it** hits a non-Ø (with no room for a 2-bit color pixel) directly underneath the starting point.

7) In small or narrow segments put the cursor at the upper right corner of the byte and hit PDL button #Ø and if necessary move your vertical coord. (PDL #1) downwards. The algorithm is trying to deal with finding no byte above or below the cursor that will hold a 2-bit color pixel (or 2 or 3 of them), so it will balk at times.

8) The filling program is meant to fill shapes that are black (Ø or 4) inside. If you want to change the color of a shape or segment, you'll need to carefully erase the old color. The program tells you a bit about this; but I shall do so also. (Incidentally, in using colors that are either half black (Ø or 128 bytes) or mystery colors, you can often fill filled segments with additional color, depending upon cursor vertical coordinate when filling.)

You get to use 1Ø different paintbrush heights and 8 colors (hi-res colors Ø-7). The brush paints horizontally only and it paints "from where you just were to where you now are." If you gave a quick turn of the paddle, the painting will continue until it's complete. If you want horizontal lines one dot thick, ask for a height of 1. The subroutine is not meant for vertical line drawing, but if you want to, set the height for 9 and move the vertical (#1) paddle knob slowly.

It shouldn't take you long to figure out that erasing larger areas takes color Ø and 9 or better for height. But **how** can you get a larger height and why does the program say 1-9 are the height choices, when I wrote above that 1Ø heights are possible? Well, if you're in the fill routine and hit Space Bar, you get asked for HCOLOR and then go to your painting routine with a quite large height (18).

See, the normal way of going from "fill" mode to "paintbrush" mode is by hitting 1-9 for brush height and then hitting Ø-7 for color. But Space Bar gives an even bigger height of 18.

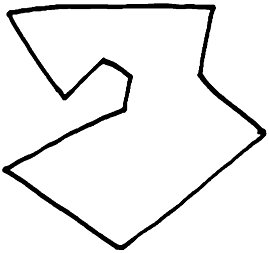
Erasing is aided by the "lift-brush" feature: if you hold down PDL #1 button and move the paddles, you get to move around **without** painting. I recommend holding it even upon entering the paintbrush routine.

Once you've erased some or all of the color from a segment, hit Space Bar and F and RETURN to fill again (or if you want a different fill color simply hit E and RETURN after Space Bar to exit and then use option 1Ø or 17 to get a new color. Either option will store 4 fresh color-byte #s between $6 and $9 in memory. In #1Ø you hit PDL#Ø button when the cursor is on the (lower screen) solid color you want, or on the left half of the color combinations to choose from on the upper screen. The fill colors that get put into memory will be the byte you're on and the one either to the left or the right of the one you're on, and the 2 bytes **underneath** these 2 chosen ones. You'll count 4Ø color squares per line, which makes 2Ø 2-color patterns. If you hit the button while you're at byte 33, you'll get the same 2 colors as at byte 32 --- you can't get 33 and 34 or 32 and 32, but just an even plus an odd (32 and 33 or 34 and 35, etc.). The upper and lower even-column (40 byte-columns per line) byte will go into $6 and $8, and the odds go into $7 and $9. More on this program in a later chapter.

It should be noted that there are only 140 distinguishable pixels per hor. line for regular color graphics on regular color monitors, but the black and white monitor gets up to 560 point resolution, although I don't agree with Bob Bishop on this subject. He states in the FALL 1980 Apple Orchard that "there's really very little reason to ever consider doing black and white plots in any other mode but the 560 mode." My opinion is that 280 resolution is much simpler and the best mode by far for all beginners; also, many applications simply don't require the extra-fine resolution. However, it's quite true that such resolution is a plus and is of significant value in many types of applications. More on resolution in a later chapter.

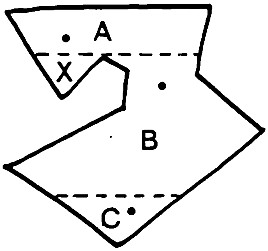
A last comment on filling with 4 of 28D, option 11:

It's probably obvious how one goes about filling a segment shaped like this:



But I find out occasionally that the "probably obvious" to some people is the "baffling dilemma of a lifetime" to others, and this graphics system is meant to be, above all else, informative. So:

(Remember the characteristics of how filling works? --- see points (5)-(8) above.)



"Seed" the segment three times. The 3 starting-cursor positions are shown. It doesn't matter where you begin. If you start at the point near the letter A in the A segment, sectors A and X will get filled, then the B sector may be filled with the B dot, and finally the C sector will be filled with the C dot. If you started with the B dot, all of A and B would be filled, C would be filled as before, but the X sector fill would have to originate right where the letter X is.

18 WHITE LINE FIX

On 28D can be found 4 of 28D, Palette. Option 18 lets you widen all the lines (vertical or diagonal, not horizontal). It does this so shapes that are too skimpy can get more definite and emphasized, or because one-line-thick shape-lines are seen as green and violet on the hi-res (**color** monitor) screen, rather than white, while thicker lines look white. Also, you may use the routine before doing color filtering (see 8 of 28D) so that shapes change colors without having various vertical lines disappear. But make sure you get this stuff --- type the following:

```
NEW
HGR: HCOLOR=1: HPLOT 10,0 TO 10,191
```

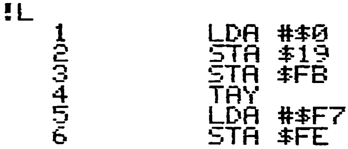
Nothing happens. Do it again with the HCOLOR of 2. A line appears! Now do it again with each color, but with hor. coords. of 11, not 10. Color 1 works well in an odd dot-column only. Violet is color 2, and it has 1010101 visible on the screen --- it's the **even** dots that are on (#0, #2, #4, #6), while green is color 1, which has 0101010 visible and only odd numbered dots are on (#1, #3, #5).

So now get one of the above lines on the screen again and type BRUN WHITELINE 1. Every single dot on the screen doubled its width (no applause, please, I blush). Now if you drew in green, draw in violet, and if you drew in violet, draw in green, and redo the above line at the same exact coordinates, ONLY THIS TIME DON'T USE THE **HGR** COMMAND. Back to a thin line. If you draw a complement on top of a visible line, it disappears. The complement line drawing masked out its opposite.

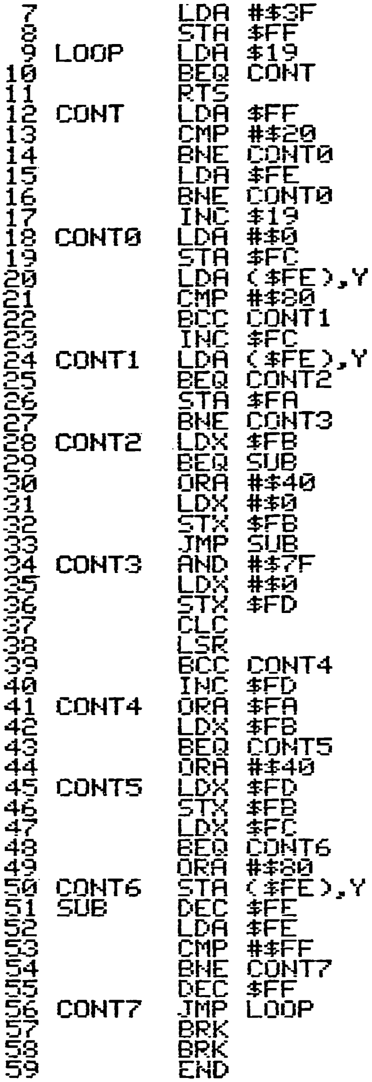
Another method of getting a white line is to draw the visible line's complement **next** to it so that the 01 and the 10 pixels add up to 11. But BRUNing the WHITELINE 1 routine is more convenient than hplotting 2497063 zillion lines and points, don't you think?

Remember to use either option 17 of 28D or option 18 or 4 of 28D before doing color filtering in 8 of 28D. You just learned why --- lines drop out otherwise --- with thick lines before filtering, you get lines slimming down but **not** vanishing. Now let's see WHITELINE (the source code for binary file WHITELINE 1):

PRINTOUT #44



```
!L
     1          LDA #$0
     2          STA $19
     3          STA $FB
     4          TAY
     5          LDA #$F7
     6          STA $FE
```



```
 7           LDA #$3F
 8           STA $FF
 9  LOOP     LDA $19
10           BEQ CONT
11           RTS
12  CONT     LDA $FF
13           CMP #$20
14           BNE CONT0
15           LDA $FE
16           BNE CONT0
17           INC $19
18  CONT0    LDA #$0
19           STA $FC
20           LDA ($FE),Y
21           CMP #$80
22           BCC CONT1
23           INC $FC
24  CONT1    LDA ($FE),Y
25           BEQ CONT2
26           STA $FA
27           BNE CONT3
28  CONT2    LDX $FB
29           BEQ SUB
30           ORA #$40
31           LDX #$0
32           STX $FB
33           JMP SUB
34  CONT3    AND #$7F
35           LDX #$0
36           STX $FD
37           CLC
38           LSR
39           BCC CONT4
40           INC $FD
41  CONT4    ORA $FA
42           LDX $FB
43           BEQ CONT5
44           ORA #$40
45  CONT5    LDX $FD
46           STX $FB
47           LDX $FC
48           BEQ CONT6
49           ORA #$80
50  CONT6    STA ($FE),Y
51  SUB      DEC $FE
52           LDA $FE
53           CMP #$FF
54           BNE CONT7
55           DEC $FF
56  CONT7    JMP LOOP
57           BRK
58           BRK
59           END
```

Line 1-4 flags and Y index are zeroed.

Line 5-8 starts fixing at page one's highest visible byte.

Line 9-11 $19 is quit flag --- if unset, continue.

Line 12-17 $FF and $FE are high and low bytes of current byte address. We check here to see if we're at page one's lowest byte address yet --- if we are, set quit flag.

Line 18-19 zero "hi bit on" flag.

Line 20-23 load current byte, see if hi bit is set --- if it is, set $FC flag.

Line 24-27 reload current byte so that BEQ instruction will work, if byte is zero go to CONT2, if byte's not Ø, store a copy of it in $FA, the temporary byte holder, and go to CONT3.

Line 28-33 check ($FB) bit transfer flag --- if it's clear then jump to SUB to decrement address and loop back again and widen the **on** bits of the next byte. If bit transfer flag is set, it means a Ø bit of the next higher address was set and needs to be widened by dumping a 1 into the highest (right-most on screen) **visible** bit of the current byte (#6). ORA #$4Ø sets bit 6 --- we set bit transfer flag to Ø now.

Line 34-36 zero hi bit so we don't shift the color bit into a visible bit with LSR. Then zero temporary bit transfer flag $FD.

Line 37-40 clear carry so BCC works, then shift right. If the Ø bit was set, there's now a set carry so we set $FD, temporary bit transfer flag. We're using temporary $FD now rather than $FB because we can't tamper with $FB setting yet because we're still going to **use** that setting.

Line 41-44 we add the current byte and the shifted byte ($FA and accumulator) and get wider shape lines, by use of ORA $FA. Now we check bit transfer flag --- if it's set we set bit 6 with ORA #$4Ø. We've transferred last byte's set Ø bit to this byte's 6 bit.

Line 45-49 dump temporary bit transfer flag's ($FD) setting into regular ($FB) bit transfer flag, and then check hi bit flag. If set, set color bit (7) with ORA #$8Ø. (Hi bit was zeroed unconditionally in line 34 and must be returned to original status, as determined by line 21-23.

Line 50 stores "fixed" current byte back in its place on hi-res page one.

Line 51-56 decrement address by one, and if low byte ($FE) goes from Ø "down" to #$FF (255), then decrement hi byte ($FF) also. Loop back and continue fixing process.

CIRCLES, ELLIPSES, POLAR GRAPHS AND SPIROGRAPHS

19

List the beginning of 2 of 28A by using F of 28B. From lines 2Ø-11Ø is an ellipse formula with the option to fill itself as it draws. It does this by plotting a point and then plotting to the other point in the ellipse that's directly horizontal of it; then it moves down and does the next line. You'll notice that these are standard formulas --- not the world's fastest. Also notice that to get a concise figure it was plotted double --- 4 half-ellipses from 4 separate starting points --- this fills all would-be gaps.

The same can be said for the circle formula in 82Ø-88Ø. In both figures there are a lot of "ifs" that slow things down. Ifs that check legal boundaries, check to see if drawing sounds were asked for, check to see if filling is required, and check to see if PDL #Ø has been pushed to signify that filling should cease --- all these are needed in my particular application.

Now, there's a program not on the menu on 28D called CIRCLE/ELLIPSE. Run this and notice that a circle is merely an ellipse with equal horizontal and vertical radii. Again we've used standard formulas.

Now, for another program not on the menu, try running SPIROGRAPH. In this program functions are defined and spirographs of any of 1Ø types are drawn (it'd be nice to color these later in 4 of 28D). These are standard polar coordinate equations --- but this is a graphics package, not a math or geometry one, so I won't go into it all. Use math books to lift any fog that has settled due to any of these programs.

20

COUNTING, TIMING, SPEEDS, DELAY LOOPS, TIMING CLICKS

Check line 5Ø in 4 of 28D. It counts the permutations performed on the color bytes. Notice that it notices when the count gets beyond 48Ø, and exits the routine at that point. (The CALL 54915 resets the stacks so you won't end up with an error due to the fact that the for-next loops are incomplete --- there are hundreds more permutations possible. Another way to prevent for-next error difficulty is to set A, B, C, and D to 6 in place of the CALL 54915 --- this will make GLEEP, my computer's name, think that all loops are completed.)

Now if you needed to monitor the counting, you could leave the screen at mixed graphics with POKE-163Ø1, Ø and then PRINT N at VTAB 21 as the permutations progressed. If you wanted to emphasize the timing aspect of your programs you could put in light clicks (P=PEEK(-16336)) or heavier clicks (2-4 of these last PEEKS).

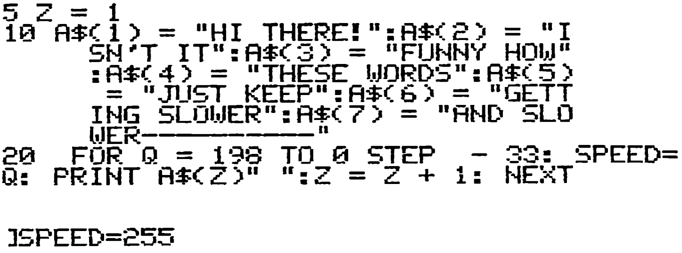
If you wanted clicks to get progressively faster then you could decrement the variable in an inserted delay loop like so:

```
5 B=3ØØ
1Ø FOR A = 1 TO 3ØØ:?"DO SOMETHING"
2Ø FOR C = 1 TO B: NEXT:B=B−1:NEXT
```

You'll find BASIC delay loops are 1 second for every 8ØØ loops, if they do nothing. I use this fact in music programs to create proportional **rests.** They're about the easiest thing you can do on a computer.

Another speed-related command is SPEED. It's used in our Chambers of Xenobia adventure to adjust the rate of speed words get printed on the screen during confrontations with monsters. It works like this:

PRINTOUT #45



```
5 Z = 1
1Ø A$(1) = "HI THERE!":A$(2) = "I
      SN'T IT":A$(3) = "FUNNY HOW"
      :A$(4) = "THESE WORDS":A$(5)
       = "JUST KEEP":A$(6) = "GETT
      ING SLOWER":A$(7) = "AND SLO
      WER————————————"
2Ø  FOR Q = 198 TO Ø STEP  - 33: SPEED=
Q: PRINT A$(Z)" ":Z = Z + 1: NEXT


 ]SPEED=255
```

I arranged to have the speed decreased so you can see that SPEED settings can cause not only slowness or fastness but changing rates of print-speed.

Delay loops in machine language are on a very different level --- a delay loop of from 2000-18000 between each of a shape sequence's erase/draw cycles wouldn't be unusual --- and animation needs to be around 11 frames/second! A normal mach. lang. delay loop would be:

```
                LDY  $FF    "highest" byte
RESTART         LDX  $FE    "lowest" byte
LOOP            DEX
                BNE  LOOP
                DEY
                BNE  RESTART
```

The numbers in $FF and $FE are multiplied together and can never have a true 1/256 (hi/lo) relationship (unless X starts at 0 and decrements 256 times before it's back to 0). (A hi/lo 16 bit number always means multiplying the high by 256 and adding the low to it.)

Clicking the speaker in machine language is a matter of reading or writing to $C030. INC $C030 or DEC $C030 are good.

Another delay mechanism is the BASIC command WAIT. It doesn't seem to be of much use in most programs, but if you want pauses that go on until certain actions are performed (like hitting the keyboard), then this can be used. See the Applesoft manual.

To have a specific delay in a machine language program, such as one that might include the above delay loops, you'd merely poke 255 with the higher value and 254 with the lower value. $FF is 255 and $FE is 254.

21 ASSEMBLY GAME PADDLE OR KEYBOARD READING

```
            CLC
            LDA  $C000
            CMP  #$80
            BCS  HITKEYBOARD
            JMP  CONTINUE
HITKEYBOARD RTS
```

The above source code will stop a routine if the keyboard's been hit. $C000 will be at least 128 dec ($80, which means hi bit set) if keyboard has been hit. $C000 and 49152 and -16384 are all the same address, and all mean **keyboard data address.** ($C010 or 49168 or -16368 is the **keyboard strobe clearing address**, and needs a 0 POKED or STAed into it to work, which then allows the next character to be read in.

```
0824-  BIT  $C061  (PDL #0 button)
0827-  BPL  $0824  (keep reading it if button unpushed)
0829-  JMP  $0803  (laser sound subroutine)
```

The above disassembly illustrates how to have a routine, such as mach. lang. laser sounds, happen only when PDL #0 is being pressed (button #1 is $C062). The noise routine starts at address $803 and ends at $823. By CALLING 2084 or JMP $824 (done in $800), we can have the laser sound only if the button is pushed.

The BIT command does several things. The action we're interested in is putting bit 7 into the N flag of the status register. A set flag indicates minus, or a set high bit (which indicates negativity in two's complement binary). BPL branches if N is 0 or hi bit **not** set, which means no button #0 hitting, at the moment. BPL means, in this case: "no one's hitting the button yet but continue to go back and keep checking."

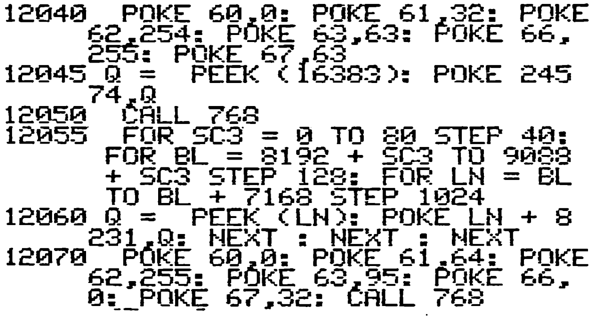
SCREEN SCROLLING 22

D of 28B will test various scrolling routines in my Superfont program. A quick 64-line upwards scroll is what's normally used in Superfont. But 5 other scrolling algorithms are available:

1) one byte leftwards, no wrap-around
2) one byte leftwards with wrap-around
3) one byte rightwards, no wrap-around
4) one byte leftwards with wrap-around
5) 8 line upward scroll

You may wish to see how these work. Let's check out the 64-byte upwards scroll, but I wish to make a comment first. As I moved lines of bytes from here to there, I calculated the line addresses mathematically. A better way to do this, if speed is the big concern (which it wasn't at the time) is YTABLE "look up address in table" methods. I'll create one such program once I've finished with the above scrolling explanations. Let's look now at a leftward scroll with wrap-around for use (as an example of the slowness of BASIC) in my Font Program, also known as **Superfont**:

PRINTOUT #46



```
12Ø4Ø  POKE 6Ø,Ø: POKE 61,32: POKE
     62,254: POKE 63,63: POKE 66,
     255: POKE 67,63
12Ø45 Q =  PEEK (16383): POKE 245
     74,Q
12Ø5Ø  CALL 768
12Ø55  FOR SC3 = Ø TO 8Ø STEP 4Ø:
      FOR BL = 8192 + SC3 TO 9Ø88
      + SC3 STEP 128: FOR LN = BL
      TO BL + 7168 STEP 1Ø24
12Ø6Ø Q =  PEEK (LN): POKE LN + 8
     231,Q: NEXT : NEXT : NEXT
12Ø7Ø  POKE 6Ø,Ø: POKE 61,64: POKE
     62,255: POKE 63,95: POKE 66,
     Ø: POKE 67,32: CALL 768
```

What does line 12Ø4Ø mean? See page 59 of your white Apple Reference Manual, also page 167 (FE2C-FE35), page 165 (FCB4-FCC8) and page 155 (55-58,61-62). What we're seeing here is memory-move algorithms. (dest) < (start). (end)M means copy from "start" to "end" and put it into "dest." In the monitor routines listed listed above, (A1L,A1H) is the lo/hi of the "start," (A2L,A2H) is the lo/hi of the "end," and (A4L,A4H) is the lo/hi of "dest." The addresses for the moving are ($3C,$3D) start, ($3E,$3F) end, ($42,$43) dest. Or start with start in 6Ø (lo) and 61 (hi), end in 62 (lo) and 63 (hi) and move to 66 (lo) and 67 (hi). Now look at line 12Ø4Ø again.

Obviously, we move the whole hi-res page one somewhere. Lo of 0 and hi of 32 (dec) is $2000 since hex of 32 is $20. For the end byte, 63 hi and 254 lo are. 2 bytes before $4000, (the last few bytes before $4000 are not visible screen bytes). The reason we take the hi-res screen one and put it up at hi-res screen two (minus one byte), is because that moves everything leftwards with wrap-around.

Understand that we moved $2000-$3FFE (page 1) to $3FFF-$5FFD, which is hi-res **page 2 minus 1 byte** which means that every visible byte has shifted left on the screen.

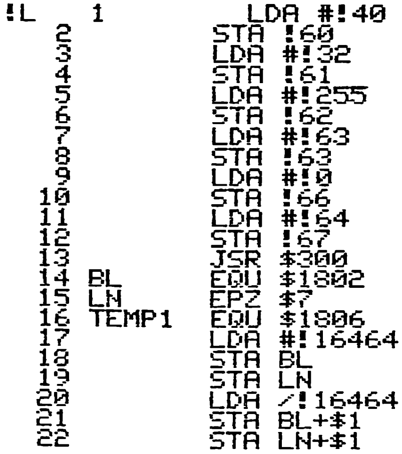
Unfortunately, the wrap-around we just got is worth less than a Nixon promise in an election year. The wrap-around we get from merely going through lines 12040-12050 is bad news --- all the bytes that jumped to the right side of the screen also jumped up 64 lines higher. So that's what lines 12055 to 12060 do --- bring up the correct wrap-around bytes from page 1 and stick them in page 2. (Line 12070 merely moves everything back from page 2 to page 1).

Why CALL768($30)? Why not CALL65068($FE2C), which is the address of the MOVE subroutine? Funny you should ask. Notice how the $FE2C-$FE2E lines are indexed with Y? Well, we don't **want** indexing, so we need Y to be 0 so we nullify that indexing. At 768 we've poked in loading Y with 0 and then jumping to 65068. This happens before all moves.

If we don't want indexing, why use (A1L),Y as an operand? Because we need the indirectness of the addressing mode --- but not the indexing. In other words, adding Y to the address is no good for our application, but indirectness, signified by parenthesis, is correct. We don't want $3C to get LDA --- A1L is $3C --- we want the address whose lo byte is given in $3C to get LDA --- hence the need for indirection.

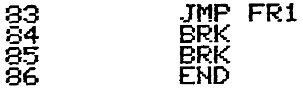
Now let's look at TEST 32, the source for the file FAST ^ 64-LN SCROLL (TEST#32):

PRINTOUT #47



```
!L    1          LDA #!40
     2          STA !60
     3          LDA #!32
     4          STA !61
     5          LDA #!255
     6          STA !62
     7          LDA #!63
     8          STA !63
     9          LDA #!0
    10          STA !66
    11          LDA #!64
    12          STA !67
    13          JSR $300
    14 BL       EQU $1802
    15 LN       EPZ $7
    16 TEMP1    EQU $1806
    17          LDA #!16464
    18          STA BL
    19          STA LN
    20          LDA /!16464
    21          STA BL+$1
    22          STA LN+$1
```

```
FR1     LDA BL
        CMP #!17361
        LDA BL+$1
        SBC /!17361
        BLT GNXT1
        LDY #!39
        LDA #!0
NX      STA !24536,Y
        DEY
        BNE NX
        LDA #!0
        STA !60
        LDA #!64
        STA !61
        LDA #!255
        STA !62
        LDA #!95
        STA !63
        LDA #!0
        STA !66
        LDA #!32
        STA !67
        JSR $300
        RTS
GNXT1   CLC
        LDA BL
        STA LN
        ADC #!7169
        STA TEMP1
        LDA BL+$1
        STA LN+$1
        ADC /!7169
        STA TEMP1+$1
FR2     LDA LN
        CMP TEMP1
        LDA LN+$1
        SBC TEMP1+$1
        BLT GNXT2
        JMP JM2
GNXT2   LDY #!39
        LDA #!0
NEXT    STA (LN),Y
        DEY
        BNE NEXT
        STA (LN),Y
JM1     CLC
        LDA LN
        ADC #!1024
        STA LN
        LDA LN+$1
        ADC /!1024
        STA LN+$1
        JMP FR2
JM2     CLC
        LDA BL
        ADC #!128
        STA BL
        LDA BL+$1
        ADC /!128
        STA BL+$1
```



```
83        JMP FR1
84        BRK
85        BRK
86        END
```

Line 1-13 move the hi-res screen to page 2, only this time all bytes are made to be 4Ø bytes **less** than they were on page one. The idea behind this can become clearer if you study page 21 of the white Apple reference manual. Notice lines 8192,8232, and 8272. You see, the hi-res screen is mapped so that if you add 4Ø to all the addresses in the top 1/3 of the screen the bytes will move down and become what's in the middle 1/3 of the screen. The same for the other 2/3 of the screen --- add or subtract 4Ø (dec.) from all addresses and you get a 64-line shift down or up, respectively.

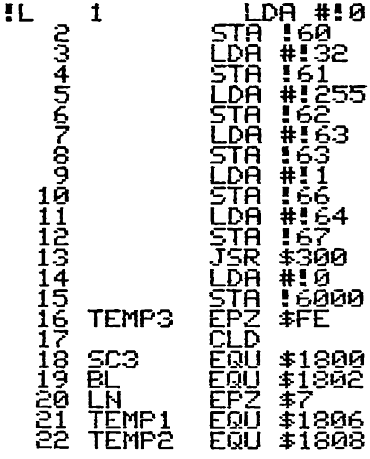
Remember that JSR $3ØØ merely zeroes Y and jumps to the 65Ø68 move routine.

Line 14-32 and **7-83** merely turn off the bytes in the lowest 1/3 of the screen, which is what we require for Superfont. It's a can of worms not worth dwelling on, but I'll mention line 62 gets a 39 (dec. is signified by !) so byte columns 39-Ø can all get zeroed. Line 7Ø gets a 1Ø24 because most lines on the screen are 1Ø24 (address #) away from the lines above and below themselves. Line 78 gets 128 because every 8th line is only 128 more than the line 8 lines above or below it (except that every 64th line is merely 4Ø more than the lines 64 lines from themselves). But why dwell on this nonsense?

Line 33-45 moves page 2 to page 1 once the bottom 1/3 of the screen is erased.

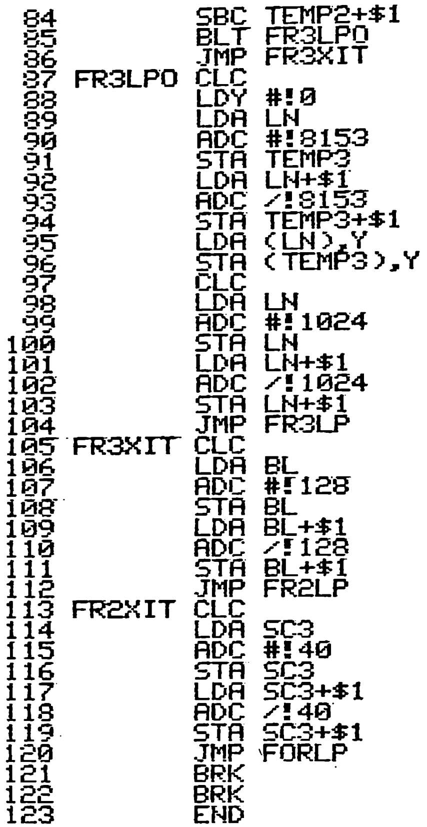
Let's check out TEST 41, the source for R > 1-BT SCROLL (TEST#41):

PRINTOUT #48



```
!L   1          LDA #!Ø
     2          STA !6Ø
     3          LDA #!32
     4          STA !61
     5          LDA #!255
     6          STA !62
     7          LDA #!63
     8          STA !63
     9          LDA #!1
    1Ø          STA !66
    11          LDA #!64
    12          STA !67
    13          JSR $3ØØ
    14          LDA #!Ø
    15          STA !6ØØØ
    16 TEMP3    EPZ $FE
    17          CLD
    18 SC3      EQU $18ØØ
    19 BL       EQU $18Ø2
    2Ø LN       EPZ $7
    21 TEMP1    EQU $18Ø6
    22 TEMP2    EQU $18Ø8
```

```
23              LDA  #!0
24              STA  $1800
25              LDA  #!8231
26              STA  $1802
27              LDA  #!0
28              STA  $1801
29              LDA  /!8231
30              STA  $1803
31  FORLP       LDA  SC3
32              CMP  #!81
33              LDA  SC3+$1
34              SBC  /!81
35              BLT  FORLPO
36              JSR  $F3F2
37              LDA  #!0
38              STA  !60
39              LDA  #!64
40              STA  !61
41              LDA  #!255
42              STA  !62
43              LDA  #!95
44              STA  !63
45              LDA  #!0
46              STA  !66
47              LDA  #!32
48              STA  !67
49              JSR  $300
50              RTS
51  FORLPO      CLC
52              LDA  SC3
53              ADC  #!9128
54              STA  TEMP1
55              LDA  SC3+$1
56              ADC  /!9128
57              STA  TEMP1+$1
58              LDA  SC3
59              ADC  #!8231
60              STA  BL
61              LDA  SC3+$1
62              ADC  /!8231
63              STA  BL+$1
64  FR2LP       LDA  BL
65              CMP  TEMP1
66              LDA  BL+$1
67              SBC  TEMP1+$1
68              BLT  FR2LPO
69              JMP  FR2XIT
70  FR2LPO      CLC
71              LDA  BL
72              STA  LN
73              LDA  BL+$1
74              STA  LN+$1
75              LDA  BL
76              ADC  #!7169
77              STA  TEMP2
78              LDA  BL+$1
79              ADC  /!7169
80              STA  TEMP2+$1
81  FR3LP       LDA  LN
82              CMP  TEMP2
83              LDA  LN+$1
```



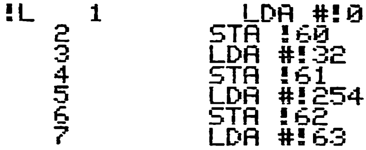
```
84          SBC TEMP2+$1
85          BLT FR3LPO
86          JMP FR3XIT
87 FR3LPO   CLC
88          LDY #!Ø
89          LDA LN
9Ø          ADC #!8153
91          STA TEMP3
92          LDA LN+$1
93          ADC /!8153
94          STA TEMP3+$1
95          LDA (LN),Y
96          STA (TEMP3),Y
97          CLC
98          LDA LN
99          ADC #!1Ø24
1ØØ         STA LN
1Ø1         LDA LN+$1
1Ø2         ADC /!1Ø24
1Ø3         STA LN+$1
1Ø4         JMP FR3LP
1Ø5 FR3XIT  CLC
1Ø6         LDA BL
1Ø7         ADC #!128
1Ø8         STA BL
1Ø9         LDA BL+$1
11Ø         ADC /!128
111         STA BL+$1
112         JMP FR2LP
113 FR2XIT  CLC
114         LDA SC3
115         ADC #!4Ø
116         STA SC3
117         LDA SC3+$1
118         ADC /!4Ø
119         STA SC3+$1
12Ø         JMP FORLP
121         BRK
122         BRK
123         END
```

From **lines 1-13** we move page 1 to page 2 plus 1, which moves everything right 7 dots (one byte) and lines 37-49 put page 2 back in page 1, but only after page one gets erased, with JSR $F3F2, which is like CALL62450. This program **has** wrap-around.

The rest of the program merely goes through a can of worms getting page one's last bytes and bringing them up to page 2's zero byte column so that wrap-around is effected.

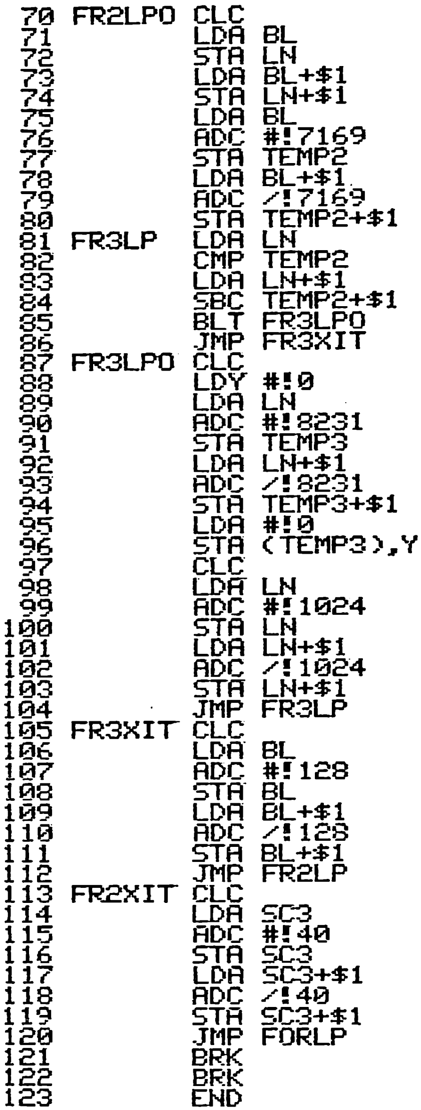
Now let's see TEST #43, which is the source for LNW < 1-BT SCROLL (TEST#43):

PRINTOUT #49



```
!L   1        LDA #!Ø
     2        STA !6Ø
     3        LDA #!32
     4        STA !61
     5        LDA #!254
     6        STA !62
     7        LDA #!63
```

```
        STA !63
        LDA #!255
        STA !66
        LDA #!63
        STA !67
        JSR $300
        LDA $3FFF
        STA $5FFE
TEMP3   EPZ $FE
        CLD
SC3     EQU $1800
BL      EQU $1802
LN      EPZ $7
TEMP1   EQU $1806
TEMP2   EQU $1808
        LDA #!0
        STA $1800
        LDA #!8192
        STA $1802
        LDA #!0
        STA $1801
        LDA /!8192
        STA $1803
FORLP   LDA SC3
        CMP #!81
        LDA SC3+$1
        SBC /!81
        BLT FORLPO
        JSR $F3F2
        LDA #!0
        STA !60
        LDA #!64
        STA !61
        LDA #!255
        STA !62
        LDA #!95
        STA !63
        LDA #!0
        STA !66
        LDA #!32
        STA !67
        JSR $300
        RTS
FORLPO  CLC
        LDA SC3
        ADC #!9089
        STA TEMP1
        LDA SC3+$1
        ADC /!9089
        STA TEMP1+$1
        LDA SC3
        ADC #!8192
        STA BL
        LDA SC3+$1
        ADC /!8192
        STA BL+$1
FR2LP   LDA BL
        CMP TEMP1
        LDA BL+$1
        SBC TEMP1+$1
        BLT FR2LPO
        JMP FR2XIT
```



```
70  FR2LPO CLC
71         LDA BL
72         STA LN
73         LDA BL+$1
74         STA LN+$1
75         LDA BL
76         ADC #!7169
77         STA TEMP2
78         LDA BL+$1
79         ADC /!7169
80         STA TEMP2+$1
81  FR3LP  LDA LN
82         CMP TEMP2
83         LDA LN+$1
84         SBC TEMP2+$1
85         BLT FR3LPO
86         JMP FR3XIT
87  FR3LPO CLC
88         LDY #!0
89         LDA LN
90         ADC #!8231
91         STA TEMP3
92         LDA LN+$1
93         ADC /!8231
94         STA TEMP3+$1
95         LDA #!0
96         STA (TEMP3),Y
97         CLC
98         LDA LN
99         ADC #!1024
100        STA LN
101        LDA LN+$1
102        ADC /!1024
103        STA LN+$1
104        JMP FR3LP
105 FR3XIT CLC
106        LDA BL
107        ADC #!128
108        STA BL
109        LDA BL+$1
110        ADC /!128
111        STA BL+$1
112        JMP FR2LP
113 FR2XIT CLC
114        LDA SC3
115        ADC #!40
116        STA SC3
117        LDA SC3+$1
118        ADC /!40
119        STA SC3+$1
120        JMP FORLP
121        BRK
122        BRK
123        END
```

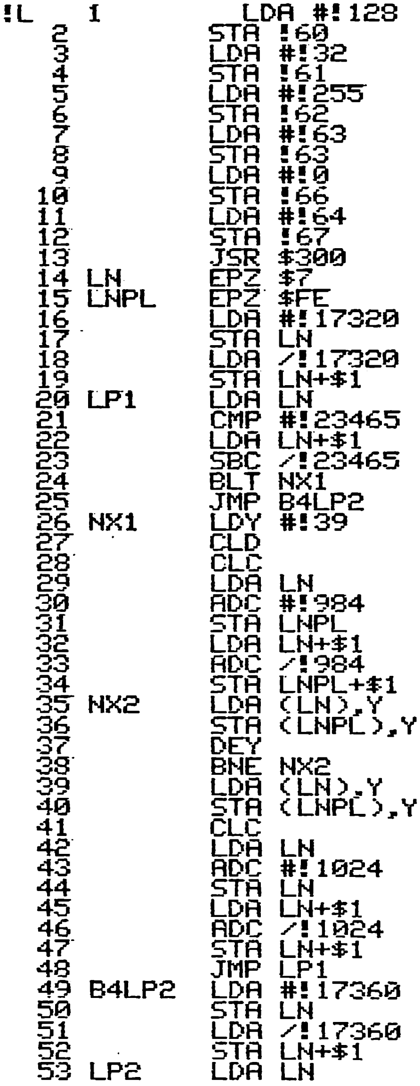
Line 1-15 puts page 1 into page 2 minus one byte, which shifts everything left.

Line 36-49 erases page 1 and puts page 2 back in page 1. This program has no wrap-around.

The rest of the program zeroes the goofy wrap-around that has occured, turning all such bytes to zero since they're 64 lines too high. See earlier scroll-program discussions.

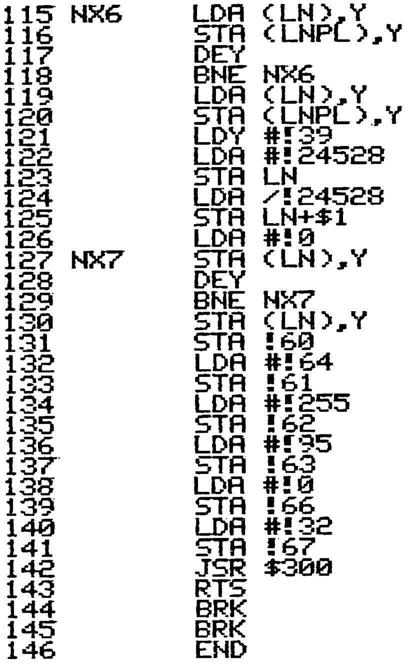
Now let's see TEST #39, which is the source for ^ 8-LN SCROLL (TEST#39):

PRINTOUT #50



```
!L   1         LDA #!128
     2         STA !60
     3         LDA #!32
     4         STA !61
     5         LDA #!255
     6         STA !62
     7         LDA #!63
     8         STA !63
     9         LDA #!0
    10         STA !66
    11         LDA #!64
    12         STA !67
    13         JSR $300
    14 LN      EPZ $7
    15 LNPL    EPZ $FE
    16         LDA #!17320
    17         STA LN
    18         LDA /!17320
    19         STA LN+$1
    20 LP1     LDA LN
    21         CMP #!23465
    22         LDA LN+$1
    23         SBC /!23465
    24         BLT NX1
    25         JMP B4LP2
    26 NX1     LDY #!39
    27         CLD
    28         CLC
    29         LDA LN
    30         ADC #!984
    31         STA LNPL
    32         LDA LN+$1
    33         ADC /!984
    34         STA LNPL+$1
    35 NX2     LDA (LN),Y
    36         STA (LNPL),Y
    37         DEY
    38         BNE NX2
    39         LDA (LN),Y
    40         STA (LNPL),Y
    41         CLC
    42         LDA LN
    43         ADC #!1024
    44         STA LN
    45         LDA LN+$1
    46         ADC /!1024
    47         STA LN+$1
    48         JMP LP1
    49 B4LP2   LDA #!17360
    50         STA LN
    51         LDA /!17360
    52         STA LN+$1
    53 LP2     LDA LN
```

```
        CMP #!23505
        LDA LN+$1
        SBC /!23505
        BLT NX3
        JMP B4LP3
NX3     LDY #!39
        CLC
        LDA LN
        ADC #!984
        STA LNPL
        LDA LN+$1
        ADC /!984
        STA LNPL+$1
NX4     LDA (LN),Y
        STA (LNPL),Y
        LDA #!0
        STA (LN),Y
        DEY
        BNE NX4
        LDA (LN),Y
        STA (LNPL),Y
        LDA #!0
        STA (LN),Y
        CLC
        LDA LN
        ADC #!1024
        STA LN
        LDA LN+$1
        ADC /!1024
        STA LN+$1
        JMP LP2
B4LP3   LDY #!39
        LDA #!8232
        STA LN
        LDA /!8232
        STA LN+$1
        CLC
        LDA LN
        ADC #!9048
        STA LNPL
        LDA LN+$1
        ADC /!9048
        STA LNPL+$1
NX5     LDA (LN),Y
        STA (LNPL),Y
        DEY
        BNE NX5
        LDA (LN),Y
        STA (LNPL),Y
        LDY #!39
        LDA #!8272
        STA LN
        LDA /!8272
        STA LN+$1
        CLC
        LDA LN
        ADC #!9048
        STA LNPL
        LDA LN+$1
        ADC /!9048
        STA LNPL+$1
```



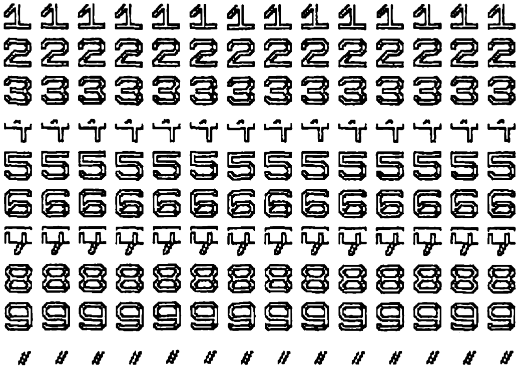
```
115 NX6      LDA (LN),Y
116          STA (LNPL),Y
117          DEY
118          BNE NX6
119          LDA (LN),Y
120          STA (LNPL),Y
121          LDY #!39
122          LDA #!24528
123          STA LN
124          LDA /!24528
125          STA LN+$1
126          LDA #!0
127 NX7      STA (LN),Y
128          DEY
129          BNE NX7
130          STA (LN),Y
131          STA !60
132          LDA #!64
133          STA !61
134          LDA #!255
135          STA !62
136          LDA #!35
137          STA !63
138          LDA #!0
139          STA !66
140          LDA #!32
141          STA !67
142          JSR $300
143          RTS
144          BRK
145          BRK
146          END
```

Lines 8 apart are 128 off, address-wise, so **line 1-13** puts page 1 into page 2, only all bytes are made to be 128 less than they were, which moves them up 8 lines.

Line 131-142 moves page 2 to page 1.

The rest of the program gets a juggling problem. See the picture below:

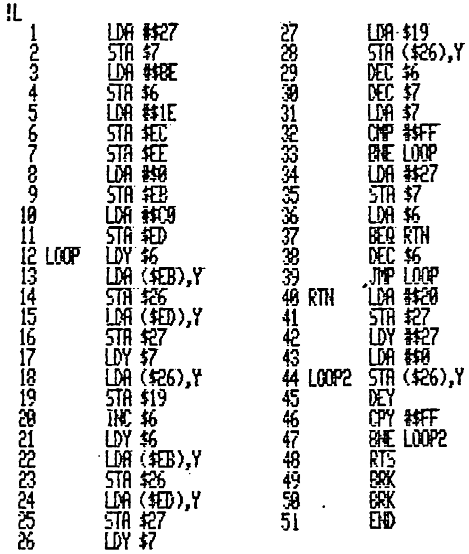
PRINTOUT #51



Notice that adding 128 to the 8 lines from 9Ø88 to 9Ø88+7168 and from 9128 to 9128+7168 does an unwanted juggling act. This program is a bit long because it's unjuggling. Notice the tops of the ones are in the broken fours and the tops of the fours are in the demolished sevens and the rest of the sevens are where a black screen ought to be. (This picture is of page 2 after that first move happened; I hope you see that all numbers were in perfect shape in page 1, and the picture illustrates the move's effects.)

Now we'll see how a table look-up scroll, 1 line upwards, would be. Here's UPSCRL and this is the source for UPSCRL1:

PRINTOUT #52



```
!L
  1          LDA #$27         27          LDA $19
  2          STA $7           28          STA ($26),Y
  3          LDA #$BE         29          DEC $6
  4          STA $6           30          DEC $7
  5          LDA #$1E         31          LDA $7
  6          STA $EC          32          CMP #$FF
  7          STA $EE          33          BNE LOOP
  8          LDA #$0          34          LDA #$27
  9          STA $EB          35          STA $7
 10          LDA #$C0         36          LDA $6
 11          STA $ED          37          BEQ RTN
 12 LOOP     LDY $6           38          DEC $6
 13          LDA ($EB),Y      39          JMP LOOP
 14          STA $26          40 RTN      LDA #$20
 15          LDA ($ED),Y      41          STA $27
 16          STA $27          42          LDY #$27
 17          LDY $7           43          LDA #$0
 18          LDA ($26),Y      44 LOOP2    STA ($26),Y
 19          STA $19          45          DEY
 20          INC $6           46          CPY #$FF
 21          LDY $6           47          BNE LOOP2
 22          LDA ($EB),Y      48          RTS
 23          STA $26          49          BRK
 24          LDA ($ED),Y      50          BRK
 25          STA $27          51          END
 26          LDY $7
```

You'll find this program on 28D. Make sure POKES 1Ø3,1 and 1Ø4,64 and 16384,Ø are in before RUNning the BASIC driver program for UPSCRL1, called SCROLL UP. You'll have to stop after a few seconds and load in disk 28C, as SCROLL UP needs to BLOAD YTABLE, the "look-up" program (more on YTABLE later in the manual):

Line 1-4 puts 39 in $7, the hor. byte holder; and 19Ø in $6, the vert. coord. # holder. We're starting at the lower right-hand corner.

Line 5-11 loads the YTABLE addresses into the proper addresses. $EB, $EC hold low/hi for the low byte of YTABLE's vert. addresses ---this table section is at address $1EØØ and gets indexed with displacements up to 191, which takes it up to $1EBF. $ED, $EF hold lo/hi for the high byte of YTABLE's vert. addresses --- this table section is at address $1ECØ and is indexed similarly to it extends 191 displacements up to $1F7F.

Line 12-14 puts 19Ø into Y and finds the address of the vert. low byte of line 19Ø and puts it in $26.

Line 15-16 does the same for the high byte of the address of line 19Ø --- finds it in the table and puts it in $27.

Line 17-19 puts 39 in Y, the hor. byte indexer (temporarily), and then loads the screen byte (from the 39th and last byte of line 19Ø) into $19.

Line 2Ø-25 here you'll see why we didn't start at line 191, the **bottom** line; we increment $6 from 19Ø to 191 and get the table address of that line and put it in $26, $27 (lo/hi). We'll be moving line 19Ø bytes onto line 191 bytes. If we wanted any wrap-around we'd store the line 191 bytes first and load them at the top at the end of the subroutine rather than loading Ø's, in lines 4Ø-48.

Line 26-28 loads line 19Ø's last byte onto line 191's last byte, thereby replacing it.

Line 29 decrements vert. coord. holder back to 19Ø.

Line 3Ø-33 moves indexer one byte left to byte #38, and then sees if our last byte was #Ø; if it was, then the decrementing at line 3Ø will have turned $7 into $FF, two's complement for -1. If the last byte hit the screen's left edge, we go on to line 34, otherwise we continue moving bytes down by going back to line 12.

Line 34-35 if a line is all moved we put 39 back into our indexer holder.

Line 36-39 we check here to see if we're at line Ø --- if we're not we go to line 38-39 and move up a line and loop back to line 12 again.

Line 4Ø-43 here we put $2Ø into our high byte of our vert. address holder, putting us at line Ø, since our low byte will already have reached Ø (see page 21 of the reference manual again). Then we put 39 into the Y indexer ($7 is unneeded now) and Ø into the accumulator --- we'll be turning bytes to black.

Line 44-48 we stick Ø in the right-most byte and then decrement the index register, see if our last displacement was Ø, and continue zeroing bytes until completed.

Notice how the entire screen stretches a bit, not in height, but in size of any smaller part, as **ripples** of byte moving wash up the screen --- it almost looks like water! To avoid this use screen flipping --- see other chapters.

Notice also that if all that was on the screen was an 1/8-screen block-shape, the algorithms in 5 or 2 of 28B will move the block-shape as if it were scrolling, without need to scroll the entire screen.

23

MAKING AN INTEGER MACHINE LANGUAGE PROGRAM WORK RIGHT FROM APPLESOFT

What is meant by an "Integer machine language program"? It's one that will run in Integer BASIC but not Applesoft.

Why won't machine language programs work in conjunction with either language? Many will work fine --- but others goof. Why? Because the zero page addresses used will be in harmony with Integer Basic but not Applesoft. See page 74-75 of your white reference manual. Here is a list of pretty decent zero page addresses to use when the driver program for a machine language routine is in Applesoft (all are in hex): ($)6, 7, 8, 9, 19, 1A, 1B, 1D, 1E, 1F, CE, CF, D7, E3, EB, EC, ED, EE, EF, FA, FB, FC, FD, FE, FF, and **sometimes** the last few $DØ line ones, DC-DF.

If you've a problem with a mach. lang. program, look at the zero page address usage. If you see LDA $1, STA $5, or other no-no's, then you might substitute from the above "O.K. list". Make real sure you change **all** $1's to $6's or you'll really goof. Also make sure that if bytes have been used sequentially for a reason that you do the same. If someone is using index displacements within page Ø, you'll need to move things from $Ø-$5 up to $FA-$FF. If longer indexing is used, you'll have to use addresses such as $3ØØ-$3CF (**don't** use $3DØ on up!).

Notice that ($26), Y actually addresses both $26(lo) and $27(hi) so consecutive usage of such zero page address utilization is quite essential.

The above advice will cure most Integer/Applesoft clashes in machine lang. addresses.

SOUNDS, NOISES, VIOLINS, SONGS, ETC. 24

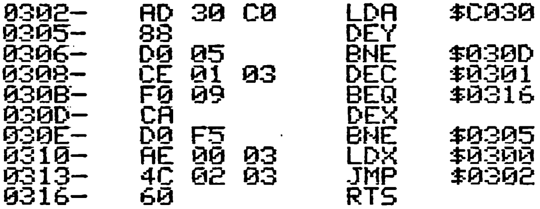
24A. TONE ROUTINES/SONGS

Run 28C, program 3 --- Tone Routine. After hearing the Close Encounters Theme, reset and list the program on your monitor. GOSUB 61000 loads the tone routine. After that, all you need do is give D,(duration) of 1-255 and P (pitch) of 1-255 and then GOSUB 60000 and your notes will be played as you want them to be. Line 60000 merely POKES your duration and pitch values into the routine, for use. A more memory-thrifty way to do a tune is with data statements:

```
          400 for Q=1 to 5:READ D,P: GOSUB 60000:
NEXT
 500DATA 140, 114, 140, 102,140, 128, 140, 255, 255, 172
```

What is a tone routine about? Here:

PRINTOUT #53



```
0302-   AD 30 C0    LDA   $C030
0305-   88          DEY
0306-   D0 05       BNE   $030D
0308-   CE 01 03    DEC   $0301
030B-   F0 09       BEQ   $0316
030D-   CA          DEX
030E-   D0 F5       BNE   $0305
0310-   AE 00 03    LDX   $0300
0313-   4C 02 03    JMP   $0302
0316-   60          RTS
```

Addressing the speaker address ($C030) toggles the speaker. The pitch, P, is POKED into $300 and the duration, D, is POKED into $301. In $308-$30C the duration # gets decremented and the routine RTS's if we've reached 0, so you can see why a small # here will get a quick note. In $30D-$30F the pitch # gets decremented and at $310-$315 it gets loaded back in and you get sent back to the speaker toggling, so if this # is small, the speaker will get toggled often and you'll get a high note --- a larger # will get a lower note, since pitch depends on frequency and frequency of speaker toggling depends upon how often X is brought down from pitch # to 0.

24B. SOUND/NOISE GENERATING ROUTINES

Okay, RE-BOOT the disk 28D and reset RUN AMPERSOUND after the disk stops spinning. Then go through and look at the way sounds are done. The variables in the FOR-NEXT loops tend to relate to the

Ampersand variables. In line 20 we're looping with I and K, and TK and TI are the Ampersand variables. That's important --- sounds will be quite different if you don't follow this format for at least parts of loops or some loops.

There isn't space to go into all the Ampersand potentials here. I suggest Volume 1 of the Nibble Express, pgs. 123-134, or Nibble, Volume 2, No. 4, pgs. 25-27, if you like this Ampersand stuff. (Or try pgs. 26-30 of Jan. 1981, CALL APPLE.)

In brief, Ampersands jump you to $3F5 (when Applesoft finds one in a BASIC program). Here you put a JMP command, perhaps to start at $300 running a mach. lang. routine. POKE1013,76:POKE1014, 0:POKE1015,3 will do this JMP $300 insertion at the $3F5 Ampersand address. The characters after an Ampersand are interpreted by the routine in $300+ and various mach. lang. routines may be jumped to as a result.

A mach. lang. program, BLOADed at $300 with a length of $42, and called AMPERNOISE will be loaded into memory to supply the Amper-Interpreting routine.

Another sound routine is called NOISES, and is found at A 5625,L114.

Run 2 of 28C and let the disk spin until it finally gives an input that stays forever. Now type GOSUB1 and RETURN. Go from 1 to 41. All it took to get these GOSUBS to where they function is to BLOAD NOISES. Lines 60-66 must be in the program. The variables in these lines are LE for length, FI for filter, DE for change, PI for pitch. You may change them to your heart's content to create new sounds of your own.

The gosubs from 1 to 21 are noise routines, using gosubs 60,64,66 --- and gosubs 22-41 are tone routines and use gosub 62. The noise routines use a different part of the NOISES file than the tone routines. The noise routines don't have pitches (PI), they use filter (FI) --- the filtered sounds are white noises, not tones.

It's easy to make the sounds of a steam locomotive by gosubing a long hiss and then using the short hisses with smaller and smaller delay loops and then once the "train" is at full steam, at about 6 hisses per second, keep the delays constant.

If you find a sound that's right for you but is too long or short, merely vary the LE (length).

24C. VIOLIN SOUNDS/SONGS

Run 2 of 28C until you get to the part where the violin song occurs. The song was made possible by 3 things:

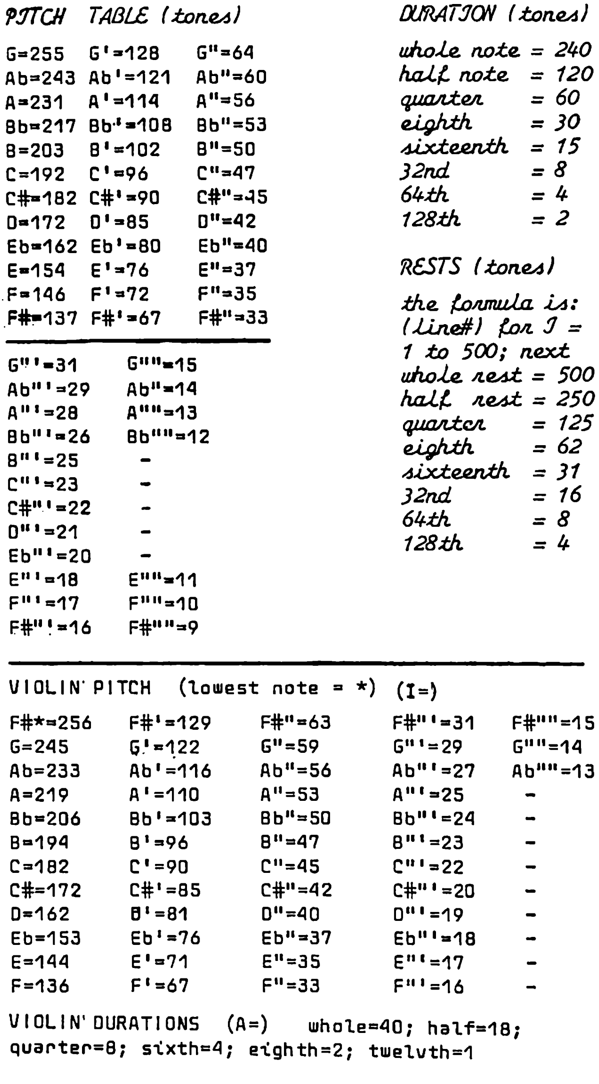
1) line 41000 read the DATA statements containing the pitches and durations for the song; I=pitch and A=duration.

2) line 42 calculated the correct values to POKE into addresses to be used by VIOLIN, and it did the POKING.

3) VIOLIN was BLOADed into memory at A7424,L66.

This is all you need to make violin tunes. Pitches from 200-256 sound best. The effect used in the sample game was to play a note, and then using the same duration, cut the pitch in half and play it again --- that's where the I/2 came in, in line 41000. A musical methodology you might try is to keep the duration (A) at 1 and perform mathematical tricks to get arpeggios, which are chords played one note at a time, up the scale. Here's a chart to help you with your creations:

PRINTOUT #54



PITCH TABLE (tones)

G=255	G'=128	G"=64
Ab=243	Ab'=121	Ab"=60
A=231	A'=114	A"=56
Bb=217	Bb'=108	Bb"=53
B=203	B'=102	B"=50
C=192	C'=96	C"=47
C#=182	C#'=90	C#"=45
D=172	D'=85	D"=42
Eb=162	Eb'=80	Eb"=40
E=154	E'=76	E"=37
F=146	F'=72	F"=35
F#=137	F#'=67	F#"=33

G"'=31	G""=15
Ab"'=29	Ab"=14
A"'=28	A""=13
Bb"'=26	Bb""=12
B"'=25	-
C"'=23	-
C#"'=22	-
D"'=21	-
Eb"'=20	-
E"'=18	E""=11
F"'=17	F""=10
F#"'=16	F#""=9

DURATION (tones)

whole note	= 240
half note	= 120
quarter	= 60
eighth	= 30
sixteenth	= 15
32nd	= 8
64th	= 4
128th	= 2

RESTS (tones)

the formula is:
(line#) for I =
1 to 500; next

whole rest	= 500
half rest	= 250
quarter	= 125
eighth	= 62
sixteenth	= 31
32nd	= 16
64th	= 8
128th	= 4

VIOLIN PITCH (lowest note = *) (I=)

F#*=256	F#'=129	F#"=63	F#"'=31	F#""=15
G=245	G'=122	G"=59	G"'=29	G""=14
Ab=233	Ab'=116	Ab"=56	Ab"'=27	Ab""=13
A=219	A'=110	A"=53	A"'=25	-
Bb=206	Bb'=103	Bb"=50	Bb"'=24	-
B=194	B'=96	B"=47	B"'=23	-
C=182	C'=90	C"=45	C"'=22	-
C#=172	C#'=85	C#"=42	C#"'=20	-
D=162	D'=81	D"=40	D"'=19	-
Eb=153	Eb'=76	Eb"=37	Eb"'=18	-
E=144	E'=71	E"=35	E"'=17	-
F=136	F'=67	F"=33	F"'=16	-

VIOLIN DURATIONS (A=) whole=40; half=18;
quarter=8; sixth=4; eighth=2; twelvth=1

Notice that if you divide a pitch in half, the result is a note an octave higher with the same name --- this 8th, as you might call it, sounds pleasing, and it's what I/2 created.

Also notice that only one note can be played at one time without either using a speaker attached to the cassette output or more hardware.

Now, the chart shows tone routine values (the top 60% of the chart) and violin values. Notice that the durations of notes and rests change in a linear fashion for the tone routine (the one from Chapter 24A) but the violin durations have a rather more logrithmic relationship.

If you don't understand music (I am also a musician) you may have trouble writing songs and figuring out notes --- but don't worry: **almost everyone** has musically-inclined friends who will at least show you how to figure out what note is what on sheet music, so you can find the notes' corresponding pitch values on my charts. The notes were figured according to how the pitches they create correspond to my guitar --- an easy method.

Well, this manual isn't about music so I'll let enough be enough and go on --- but I may as well give a couple of hints about arpeggios first.

Playing a 1st, 3rd, 5th, and 8th makes a chord if simultaneous and a rudimentary arpeggio if sequential. Notice in the tone routine pitch table that G is 255, B is 203, D is 172, and G' (the ' is read "prime" and means an octave higher) is 128. The G's are a 2/1 ration, but what about the 3rd (B) and 5th (D)?

If you used I values of I, (I*.796), (I*.6745), (I*.5), you'd get your chord/arpeggio. A 4th, in case you know what that is, is (I*.75). If each successive duration was multiplied by .75 you'd get an arpeggio of 4ths, which would sound like something from the Blood, Sweat, and Tears band. I=(I*.75) and then I=(I*.75) again --- etc.

24D. MUSIC (WRITE:RECORD:PLAY)

In this program, 1 of 28C, you'll get to write music, play it back, record songs for posterity, or drive your pet hound crazy as a looney bird. You'll see your notes correctly named and positioned on clefs on the screen. All the instructions you'll need will be on the card that comes with this manual and/or in the program itself. Flow charts are available, and included in the Creativity Life Dynamic manual, which comes in our Creativity Tool Box --- whatever.

Try saying yes when asked if you'd like to hear a stored tune. Yes, old Fudgie made that one up --- no applause --- just throw RAM chips.

Tunes are stored in text files --- to store more than one use separate disks. The sounds you hear are from the general tone routine. All you do is play the keyboard like a piano to get notes --- "black" keys are included. You might wish to list the program and see how it's done. Notice that since things like commas don't store very nicely into text files, all notes were converted into ASCII value numbers before the text file was written.

Also notice that several notes have been set aside as duration choosing keys. These notes are in positions where there would not be any keys on a piano configuration.

24E. WORDS --- SPEECH

As this is written there are probably 2 dozen voice/speech packages under development. It was intriguing when Appletalker™ and Apple Listener™ came out (by Bob Bishop and Bill Depew and put out of SOFTAPE), but the amount of memory required per word was enormous --- 20 words per disk seemed a **bit** uneconomical! But someone had to be **1st** and the programs are good deals for the price.

In Appletalker™ voice is accepted through the casette input part of the Apple. It digitizes and stores the information in tables, which can be played back later.

Another package like Appletalker is The Voice, by Muse Software, which doesn't seem to be much improvement over Appletalker --- it just costs more.

Finally, there's ECHO II, which includes speaker, card, and software. The firmware card includes an upgraded version of the Texas Instruments TMS 5200 speech synthesizer chip. This chip reconstructs human speech when supplied with the basic human speech phonemes. You get lots better sound and lots more words, and a more dynamic word/phrase composition methodology. The weakness here is that it costs more and you need hardware, so you won't write programs that share your "heavy sounds" with the world.

24F. OUR FAVORITE SOUNDS

Each of the following are individually assembled mach. lang. sound routines that will be found on 28C:

1) Helicopter(Call5548,A$15AC,L$4C
2) Bombdrop(Call3091),A$C13,L$31
3) UFO Take-off(Call4307),A$10D3,L$B0
4) Bounce2(Call4993),A$1381,L$A5
5) Multiple Laser(Call2230),A$8B6,L$2C
6) Dive Strafing(Call5334),A$14D6,L$89
7) Outer Space2(CALL5159),A$1427,L$AE
8) Explosion(CALL 5472),A$1560,L$4B
9) Foghorn(CALL 2571),A$A0B,L$78

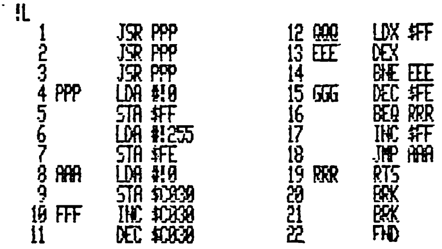
To hear them, simply type BRUN and then type the title including the parentheses and CALLS.

To save them onto your own disks type BSAVE and then **all** the info given above for each program.

To use them in your programs, BLOAD them in deferred mode early in your program and simply CALL the # in parentheses later. Make sure your program and sound routines don't reside in the same parts of memory. POKE103,1:POKE104,64:POKE16384,0 will keep them apart --- it must be done in your Hello or Menu program, not in the actual game.

Let's look at a sample sound source code. This one, MULTIPLE LASER!, is available on our Action Sounds and Hi-Res Scrolling disk, and creates the binary file MULTIPLE LASER(CALL2230):

PRINTOUT #55



```
!L
  1        JSR PPP          12 QQQ    LDX $FF
  2        JSR PPP          13 EEE    DEX
  3        JSR PPP          14        BNE EEE
  4 PPP    LDA #!0          15 GGG    DEC $FE
  5        STA $FF          16        BEQ RRR
  6        LDA #!255        17        INC $FF
  7        STA $FE          18        JMP AAA
  8 AAA    LDA #!0          19 RRR    RTS
  9        STA $C030        20        BRK
 10 FFF    INC $C030        21        BRK
 11        DEC $C030        22        END
```

Line 1-3 make the entire routine happen 3 times before the 4th and final time.

Line 4-5 put a 0 into $FF.

Line 6-7 put 255 into $FE.

Line 8-11 toggle the speaker a few times.

Line 12-14 decrement X, which is 0 now, all the way down to 0 again (256 loops). Remember that 0 decremented gives $FF, which is 255, or if you'd like to think in terms of two's complement, -1.

Line 15-16 decrement 1FE, the counter address that will see to it that 255 loops from AAA to line 18 are completed.

Line 17 is the real key to the algorithm. X got decremented down to 0 after being loaded with 0 above and decremented back to 0, but $FF was never affected. Here we change $FF from 0 to 1. This will mean that the next X decrementation will be over without looping at all. So a very high tone with few instructions between speaker toggles will result. But not for long: each time line 17 is used, the value in $FF increases, and the X decrementation loop gets longer and longer. So the sound quickly goes from a high tone to a low one --- the stereotype of "laser fire" in many games or movies.

Line 18-19 keeps you looping back to AAA until $FE reaches 0 ---then line 16 sends us to RTS in line 19.

For customizing, the various ways you might change this routine is to change the #s stored in lines 4 and 6, add or subtract more JSR PPP from the start of the program, add more delay loops between lines 11 and 12, using the Y register, or getting fancy and starting out this entire routine with a BASIC GOSUB4 (make sure NOISES is BLOADed and line 4 (from SAMPLE GAME) and line 60-66 are around) just before you CALL 2230 which activates the MULTIPLE LASER routine. I won't tell you what will happen. See for yourself.

MEMORY ORGANIZATION 25

The career of every beginning programmer is punctuated profusely with episodes of hair-pulling with respects to memory organization chaos. Programs bump into mach. lang. programs, variables and shapes tangle up, string storage or arrays eat up routines, routines bomb programs due to zero page or DOS conflicts, and hi-res pictures and animations get "totally unexplainable" white lines appearing in them.

"It must be a power surge", "bad chip", "a bug in DOS", and other such nonsense is offered in explanation, but in our heart of hearts we know that we've simply not yet figured out how all the things we do interact with each other.

There are 256 pages of memory in an Apple II Plus with 48K, and each page has 256 distinct memory locations. This means there are 256*256=65,536 bytes that can be referenced --- $0000 to $FFFF. "Page 27" starts at $2700 and ends at $27FF.

The locations from 0 to $BFFF are the 48K RAM (49,152 bytes). From $C000 to $CFFF are input/output locations such as keyboard, game paddles, etc. From $D000 to $FFFF is normal ROM, in which Applesoft and monitor locations are found. The monitor goes from $F800 to $FFFF. See Chapter 4 in your white reference manual.

Your **zero page** is used by the monitor, by Applesoft, and by DOS. There are still about 25 locations free for user mach. lang. programs, however. This is fortunate, because good fast mach. lang. programs NEED zero page adddresses!

Page 1 is the "stack".

Page 2 is the GETLN subroutine, used for getting INPUT lines.

Page 3 is free to the user, except for $3D0 to $3FF, which get special jump instructions.

Page 4-7 are the 4 text and/or lo-res graphics pages. Also peripheral firmware cards get 64 locations for temporary data storage in this range of memory.

Page 8-11 are secondary text and lo-res graphics pages, not often used or needed. Page 8-11 are free to the user, if the secondary page is not desired.

Page 12-31 ($0C-$1F) is free RAM.

Page 32-63 ($20-$3F) is free RAM **unless** you are using hi-res page 1 (HGR).

Page 64-95 ($40-$5F) is free **unless** you are using hi-res page 2 (HGR2).

Page 96-191 ($60-$BF) is free RAM, the top locations of which will often be used for Applesoft string storage, unless HIMEM has been reset.

Perhaps you can see that in most hi-res programs using only 1 screen (page one), it makes sense to POKE103, 1:POKE104,64:POKE16384,0 in Hello so your program will start at $4000 and have clear memory all the way to $9600. If you're using 2-page flipping animation, POKE103, 1:POKE104,96:POKE24576,0 in Hello so you'll start your program (that gets loaded in after Hello) at $6000. There are exceptions to this memory usage, of course, but these generalities will set you up right for 90% of hi-res using programs.

You need to be aware that if you don't change the start-of-program pointers in decimal 103 and 104, you'll always get your programs loaded in at $800. To reset your pointers to this starting point, POKE103,1:POKE10,8:POKE2048,0. You can still use the hi-res screen when your program loads here, but you're asking for trouble many times, since $800 to $1FFF is not much room, and variables and arrays get stored right above your program and it's easy for these to accumulate and begin eating graphics for breakfast. One way to preclude serious garbage collections is to have X=FRE(0) in your program in strategic locations.

Himem is something you need when you want to store tables or routines or shapes safely in high memory. High memory goes up to $9600 (unless you don't mind dumping DOS) and if you have shapes or whatever from $9000 to $95FF your correct HIMEM is HIMEM:36864, which is equal to $9000.

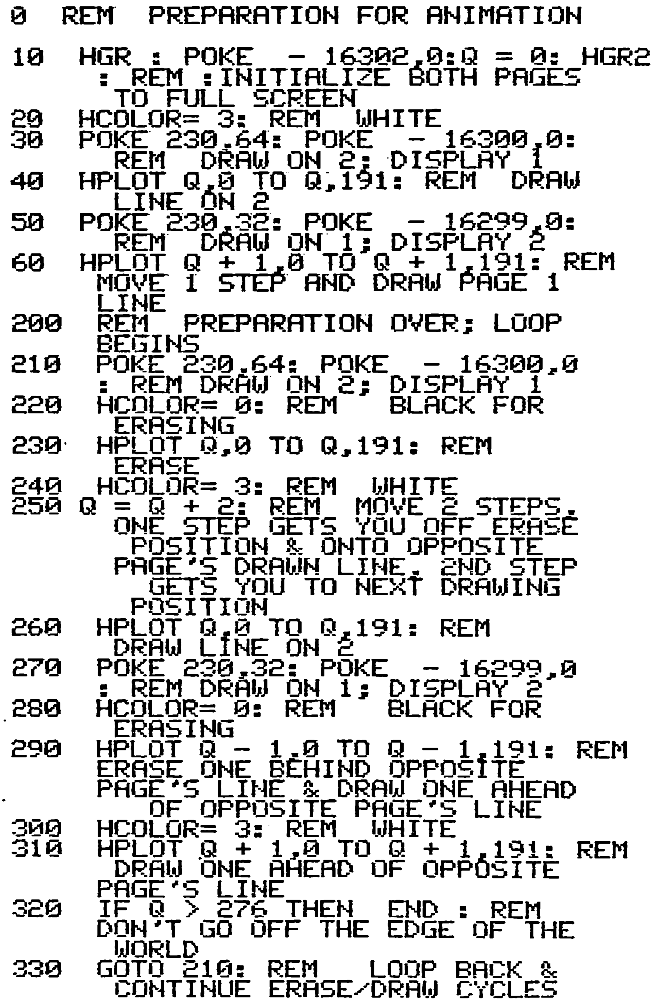
It's a good rule to keep Himem and tables and things as high in memory as you can, so you don't end up with an "out of memory" error.

Lomem is something to forget, in Applesoft, except in very unusual circumstances, since it's automatically set anyway. It sets the address to the end of the program where the variables normally begin. If you wish to move it elsewhere make sure you know what you're doing. It's the "start variables here" pointer.

These days 48K is almost a must, more is nice but less really limits your Apple's potential.

SCREEN FLIPPING 26

PRINTOUT #56



```
0  REM  PREPARATION FOR ANIMATION

10   HGR : POKE  - 16302,0:Q = 0: HGR2
       : REM :INITIALIZE BOTH PAGES
        TO FULL SCREEN
20   HCOLOR= 3: REM  WHITE
30   POKE 230,64: POKE  - 16300,0:
        REM  DRAW ON 2: DISPLAY 1
40   HPLOT Q,0 TO Q,191: REM  DRAW
        LINE ON 2
50   POKE 230,32: POKE  - 16299,0:
        REM  DRAW ON 1; DISPLAY 2
60   HPLOT Q + 1,0 TO Q + 1,191: REM
       MOVE 1 STEP AND DRAW PAGE 1
       LINE
200  REM  PREPARATION OVER; LOOP
       BEGINS
210  POKE 230,64: POKE  - 16300,0
       : REM DRAW ON 2; DISPLAY 1
220  HCOLOR= 0: REM   BLACK FOR
        ERASING
230  HPLOT Q,0 TO Q,191: REM
        ERASE
240  HCOLOR= 3: REM  WHITE
250 Q = Q + 2: REM  MOVE 2 STEPS.
        ONE STEP GETS YOU OFF ERASE
         POSITION & ONTO OPPOSITE
        PAGE'S DRAWN LINE. 2ND STEP
           GETS YOU TO NEXT DRAWING
         POSITION
260  HPLOT Q,0 TO Q,191: REM
        DRAW LINE ON 2
270  POKE 230,32: POKE  - 16299,0
       : REM DRAW ON 1; DISPLAY 2
280  HCOLOR= 0: REM   BLACK FOR
        ERASING
290  HPLOT Q - 1,0 TO Q - 1,191: REM
       ERASE ONE BEHIND OPPOSITE
       PAGE'S LINE & DRAW ONE AHEAD
           OF OPPOSITE PAGE'S LINE
300  HCOLOR= 3: REM  WHITE
310  HPLOT Q + 1,0 TO Q + 1,191: REM
        DRAW ONE AHEAD OF OPPOSITE
       PAGE'S LINE
320  IF Q > 276 THEN  END : REM
       DON'T GO OFF THE EDGE OF THE
        WORLD
330  GOTO 210: REM   LOOP BACK &
        CONTINUE ERASE/DRAW CYCLES
```

(Hit Reset and **RUN TWO-PAGE ANIMATION** on 28D)

Here is a blow-by-blow account of a 2-page BASIC animation. It's slow, and it's about as simple as 2-page flipping can get --- notice that most of the lines are me blabbing a lot, with a little Applesauce throw in to make something happen. The idea was to give a complete account of every move so ambiguity would be nil.

Think of a line being drawn at Ø (totally filling all Y coords. with that X coord. of Ø) and then at 1 and then at 2 --- etc. You **see** the drawing taking place, and the old lines would stay visible. So 2 things are necessary if a **moving** line is to be shown:

1) lines must be erased once new ones have been drawn.

2) you must make erasing and drawing something that happens out of sight, if you're going to create the illusion of one line moving rather than a lot of lines being drawn and erased.

Number one above is accomplished by erasing old lines before drawing new ones.

Number two is accomplished by switching hi-res pages so that **all drawing and erasing takes place on undisplayed pages only.**

The sequence of action will be:

draw Ø on 2 but display 1 } preparation
draw 1 on 1 but display 2 }
erase Ø/draw 2 on 2 but display 1
erase 1/draw 3 on 1 but display 2
erase 2/draw 4 on 2 but display 1
erase 3/draw 5 on 1 but display 2
ad infinitum ---

56Ø-POINT RESOLUTION 27

If the screen is 28Ø by 192, how can you plot 56Ø dots horizontally? We can't really plot half a dot, can we? What gives?

First off, convenient and easy 56Ø-point resolution is not yet here. There's too much messing around to call it easy --- however I've a feeling that 56Ø-point resolution may be everywhere in the near future. But if it isn't that's okay --- there are many applications that honestly require no more than 28Ø (black and white) points.

The maximum color resolution is 14Ø pixels, with each pixel being composed of a set of bits that can be 11,ØØ,1Ø or Ø1, and the high bit on can change the colors here by fooling the TV into thinking there's another couple of colors by changing the frequency.

What happens in 56Ø-point resolution is that we take advantage of the fact that white 2, which is HCOLOR #7, plots 1/2 dot to the right of where white 1, which is HCOLOR #3, would plot.

You see, whenever the high bit (the color bit) is on the dots in the byte shift 1/2 dot right. If you have a violet dot in location $2Ø1Ø and the byte is a 1 now (ØØØØØØ1), and then you turn on the high bit by POKE 82Ø8,129, the result is a slight rightwards shift of the dot, and on a color monitor we've gone from violet to blue. (82Ø8 is $2Ø1Ø because $2ØØØ is 8192 and $1Ø is 16 and 8192+16=82Ø8. Ø bit ON is 1 or $1 and 7 bit ON is 128 or $8Ø. If color bit (7) and Ø bit are both ON then we have 1+128=129 or $1+$8Ø=$81. To have **only** the Ø bit on, as was the status at the beginning of this paragraph, we need only to POKE82Ø8,1 or CALL-151 and do a *2Ø1Ø:Ø1).

We don't really notice half-dot shifting when we use hi-res screens, whether black or color. The only time we really notice lack of resolution is when very small shapes look funky or when diagonal lines are HPLOTTED.

So how are we going to **take advantage** of this 1/2 dot shift when the color bit is on? Well, in color work, we aren't, really --- our resolution is still 14Ø, horizontal. But in B&W we **can** do something --- though it's a bit of a hassle: (Hit Reset and **RUN RES.56Ø** on 28D)

PRINTOUT #57

```
Ø  GOTO 1Ø
1  ON Q GOTO 3,4,5
2  HCOLOR= 2:  GOTO 8:  REM  Q WAS
    Ø
3  HCOLOR= 6:  GOTO 8:  REM  Q WAS
    1
4  HCOLOR= 1:  GOTO 8:  REM  Q WAS
    2
5  HCOLOR= 5:  GOTO 8:  REM  Q WAS
    3
```

```
8  HPLOT XX% / 2,Y%
9  GOTO 50
10  HGR : HCOLOR= 3: HPLOT 70,0 TO
     60,159: REM  1ST 280 POINT R
     ESOLUTION WAY
20  FOR Y = 0 TO 159
30 XX% = 280 - Y / 8:Y% = Y
40 XT% = XX% / 4:Q = XX% - (4 * X
     T%): GOTO 1
50  NEXT
60  REM  ANOTHER WAY:
70 X = 105:XX = 159 / 20
80  FOR Y = 0 TO 159 STEP XX
90  HCOLOR= 4
100  HPLOT X + 1,Y TO X + 1,Y + (
      XX - 1)
110  HCOLOR= 7
120  HPLOT X,Y TO X,Y + (XX - 1)
125  HCOLOR= 3
130 X = X - 1:Y = Y + XX
140  HPLOT X + 1,Y TO X + 1,Y + X
      X: NEXT
```

Line 10 draws a 280 pt. res. crummy line.

Line 20-50 and **1-9** draws in 560 pt. res. and is Bob Bishop's way. **Line 70-140** draws in 560 pt. res. and is R.H. Good's way, as you Orchard readers may remember. (To do the opposite slant, change line 100 so you're adding 3 to X both times, not 1; and change line 130 so you're incrementing X, not decrementing it, and change line 140 by knocking off the two "+1"s after the "X"s.)

Let's forget the Bishop method --- too slow and crude.

The Good method alternately plots the line segments of a diagonal line in HCOLOR3 and HCOLOR7 which you know (by now) will shift the white dots 1/2 dot over so that if HCOLOR3 drew a segment on the vertical line column with the X coord. of 102, then drawing the same line in HCOLOR7 at X coord. 10 will draw it at 102 1/2, which may just be an easier way to think of it than 205 out of 560 points. (Incidentally, the HCOLOR4 black line segments get the segments uniform and clean --- that's what lines 90-100 are doing there.)

There are times when this degree of resolution would be appropriate. But don't worry about it for most applications.

LIST-OUT PROGRAMS 28

On disks 28C and 28D there are many programs you may wish to list out. The disks are not protected so you may do so. The binary files and their source codes for the animation routines of disks 28A and 28B are all found on 28C.

Programs on 28A and 28B are listable even though protected! Use programs E and F of 28B to list out anything in Applesoft on disks 28A and 28B. EXECing and such was used to make the programs listable but still protected. I won't go into the details because this is a graphics package basically.

29 COLOR FILL-IN

There are reasons why people design algorithms that fill in all bounded spaces completely, and there are reasons why people design algorithms that fill in (completely the first time): shapes like circles, squares, ellipses, rectangles, and some triangles and irregular shapes, but need 2 or more fill-initiations for certain types of irregular shapes or spaces.

An **Edu-Paint** has in mind the instantaneous (almost) filling of **any** space, even if already colored. It's very applications-minded and it executes beautifully.

However, it isn't as fast as my Palette program because it can **do so much**. My color-filler fills only black spaces (black 1 or 2) and it stops when it's found either its first all-ones byte or its first "no room for color" byte either directly above or below the original starting place. Non-Ø bytes to either **side** of the seed-byte **don't** stop it.

Because I ask less of Palette, it's much speedier. I'm told that On-Line Adventures (this isn't confirmed, but it is logical) use a Palette-like program to fill their multitudinous hi-res drawings with color. Many shapes fill completely, while others need several strategic points within the shape to be "seeded," which means you JSR to or CALL the fill routine beginning at given coords.

When I decided which type of algorithm to go for I had to choose between speed and completeness. I chose speed for these reasons:

1) the algorithm would be much simpler to explain, and this is a **learning** package above all, and I've no desire to get so complex I lose everyone except the experts.

2) I felt people would appreciate being able to use the algorithm to do the very quick picture-coloring that On-Line and others are doing (I believe they may be using their new compiler to speed up these programs). In fact, I've got fellows writing adventures for Avant-Garde Creations right now (August, 1981) who need such an algorithm, and speed would be much more appropriate than completeness. I intend to give this FILL1 algorithm (on 28D) to all who buy this package, so that the block-shapes and hplot-shapes they create (or the vector shapes they've already created with packages likes SUPER SHAPE DRAW AND ANIMATE) can be quickly and easily colored. You'll be amazed how much fun it is to color and paint with the Palette program!

3) it would take up a lot less room in memory

4) it would take less time to develop

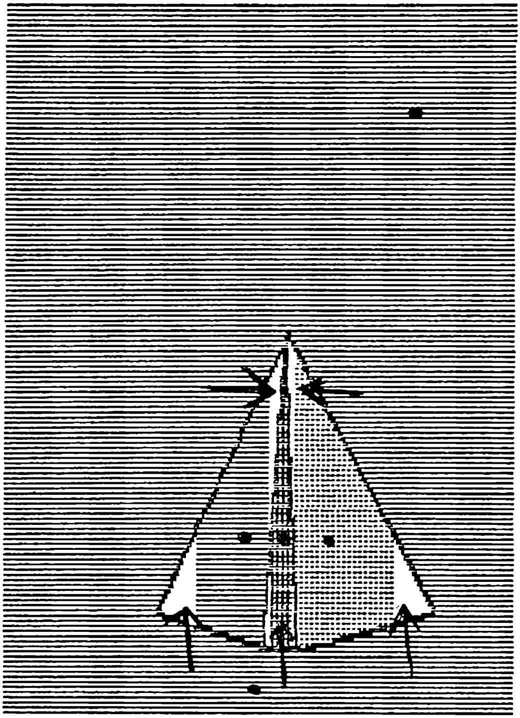
Let's look at the differences between Edu-Paint and Palette. Edu-Paint thinks in terms of 2-bit color pixels, 14Ø per line. It goes along memorizing addresses of irregular bytes above and/or below its present address and it pushes them on the stack. Every 3 1/2 pixels it has to deal with both the last bit of one byte and the first bit of the next byte (I'm talking visible bits). Once it's done a line, if it's found holes to climb through or shape nodules to go fill, it gets the starting addresses of these from the stack and goes there. In the nodule may be other nodules or off-shoots, so it may collect up lots of addresses.

As you can imagine, there's a lot involved in reading the status of all the bytes around you as you go along filling. Things can get complex. As a matter of fact, just the simple algorithm FILL1 that **doesn't** store addresses and fill everything required a flow-chart; the first time I've every actually **needed** a flow-chart in assembling. The source was 4Ø5 lines long and took only 5 days to figure out and perfect and assemble (Fudgie's 5-day Figure-Filling Fantasticon), and yet it would have taken twice that with no flow chart and up to 5 times that (?) to go for the "complete" method. I say this to encourage all of you who have heard "he took 6 months to do so and so and 2 years to do so and so" about Assembly. I too heard that stuff. There **are** programs that require such time requirements, I suppose, but in general all the rumors are pure unadulterated you know what --- these rumors are spread to scare people away so that certain people corner the market. My only advice about this is simple: don't buy it --- I didn't --- such beliefs are merely hindrances and creative barriers. Find out **for yourself** "how long things take."

The added dynamic of filling even over colored areas added to the complexity of Edu-Paint considerably, that's for sure. How do you decide if you're at a shape boundary if all the bytes you've just been traversing were already full of on and off bits rather than all off? I guess you "fix" the shapes (before they get colored) by making the lines double-thick (my WHITELINE1 program which runs from option 18 in Palette, which is 4 on 28D) and then hunt for 2 bits **on** in a row (not counting the color bit). At least, that's how a Fudge would do it. How would you?

Okay, we're going to go through FILL (the source for FILL1) very carefully, but first let's make sure we understand color filling with my FILL1 program. We've already run through it in Chapter 17, but now let's go through the Color Fill Demo, 9 of 28D, so that we can see how fun, fascinating, and easy it is to do On-Line type color filling (you may take that as a compliment, On-Line, your pioneering in the hi-res colored picture area was good for the Apple software industry). So choose 9 on 28D from the MENU and when you've run it and it says "THAT'S ALL FOLKS!", hit Reset and POKE-163Ø4,Ø and POKE-16297,Ø and look at the picture:

PRINTOUT #58

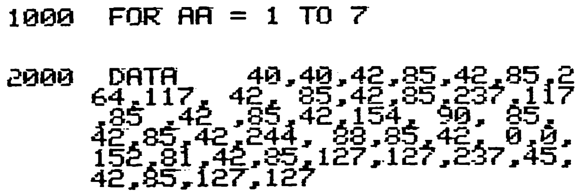


Now list the program out. You'll see that TEST O (CALL2048) was needed to draw the Hplot-shape, and T2 was the hplot-shape table that was therefore loaded, and FILL1 is the filler program that was loaded and WHITELINE1 was the optional line fixer program --- did you notice the lines getting fixed? If not do it over. (Reset and RUN).

In **line 60** is **green** getting put into the color byte addresses. See the HCOLOR chapter if this isn't clear. In **line 45** we put the hplot shape # into $7, the shape # holder.

In **line 1000-1010** we read the "seeding" coords. (at which the fill is to start) and then the color bytes' #'s. Then we GOSUB 950 for each seeding desired and do the filling. Notice that I've marked off the seeding points (with dots) on the T2 picture. They show that the seeding was done wrong inside the shape (outside the shape it was done okay). Here are the required line changes to make the seeding happen right:

PRINTOUT #59



```
1000  FOR AA = 1 TO 7

2000  DATA    40,40,42,85,42,85,2
      64,117, 42, 85,42,85,237,117
      ,85 ,42 ,85,42,154, 90, 85,
      42,85,42,244, 88,85,42, 0,0,
      152,81,42,85,127,127,237,45,
      42,85,127,127
```

Notice that I've marked off the correct seeding points with arrows. The middle section should have been seeded much further right, because the lines above and below it are diagonals going up and to the right. The wings should have been double-seeded at least (perhaps with added touches at wing-tips), which is why 2 more seedings were added to both the FOR-NEXT at 1000 and the DATA at 2000.

Another thing to watch for is mixing white 7 (hi bit on) shape lines with hi bit off colors like violet or green, or white 3 lines (hi bit off) with orange or blue (ON), or orange and green or blue and violet colors horizontally next to each other, even if separated by a line. A byte's hi bit must be either on or off!

All you need to write programs that draw pictures and color them is to use 6 DATA statements (X,Y,A,B,C,D) per seeding and the GOSUB 950, which works only if FILL1 has been BLOADED into memory. (The WHITELINE1 is up to you and seems even more valuable for color filtering (next chapter) than fill-in work.)) Of course, you need to load shapes or scenes in first so there's something to color!

Incidentally, you may use binary file hex tables in which each 4 addresses (consecutive) contain a set of color bytes. By indexing to the starting color byte of a set of 4, you can have 3 #'s per seeding rather than 6; the 2 coords. and the table index displacement.

You need only figure out the coords. of the proper seed points and the color byte numbers for each color desired and you're in business. **To get coords. while in 4 of 28D, just hit C for coords. after each fill** --- you'll even be given the color byte #'s --- jot all this down each time and you'll be all ready for **automatic** fill-in at some later date! Simple enough, right?

Now let's go tó A of 28D. Run it. **This** is the way things look when you correctly make an "instant scene" program. Color Fill Demo (9) illustrates errors in the use of color-fill routines. Bad choice of color, wrong "seeding" locations, wrong number of seeding locations, and bad mixture of high-bit-on (#4-7) and high-bit-off (#0-3) colors are shown. (Any color byte of 128 or more has high bit on. Less than 128 means high bit off.)

Good Color Fill Demo (A) illustrates a good color fill scene --- you may probably come up with better color choices, but the seeding coordinates don't have much room for improvement. Run A of 28D now and notice that in 6.7 seconds there are 89 seedings, which is over 13 per second. Notice how fast each area seeds, even large ones. This is what I had in mind when I made FILL1 do less than such do-everything routines as the one found in Edu-Paint, which is the best fill algorithm program on the market --- it not only can fill mazes and crawl through tiny holes to fill other sections --- it also can fill on top of any color.

List out A of 28D after it's done filling. (You'll have to hit Reset right as soon as the filling quits or the program will return to the menu.) In **line 11 and 12** we BLOAD 1st the picture to color and 2nd the fill routine. Data statements go from **line 20-36**. This data is coordinates and color bytes. **Line 50** has us GOSUB 1, where we read 2 coords. (hor. is X, vert. is Y) and 4 color bytes (A goes in $6, B in $7,C in $8, D in $9). This GOSUB is used only when a **new** color is being used in filling. When you use the same color for many fills in a row, you need only change the seeding coords. The color bytes will stay the same.

Line 60 is how we handle many seedings in a row with the same color (other lines up to 100 also illustrate this). We read the 2 coords. from the data and go to 950 and fill and then return for the next 2 coords. --- and so on.

Line 950 does the work. To speed up routines even a bit more you could have a line 150 that omits POKES for $6-$7, and was used everywhere except in line 1. Line 1 needs these POKES since it's dealing with new color bytes. Compilers could help the BASIC part of this fill program speed up --- but not very much: most all the action is done via machine language via FILL1.

To create this program was child's play. All I did was use 4 of 28D and color the picture I call COMPOSITE 2, and after each fill I hit C and got coord. info. which I immediately jotted down. And whenever I changed colors I jotted down all 4 color bytes immediately after the 2 coord. #'s. My jottings looked like this:

1) 37,110,128,128,42,85
2) 25,102
3) 30,96
(etc.)

When I got to a new color I again used 6 #'s rather than 2 #'s. In putting together lines like line 60, I merely counted all the 2-number data lines between the previous and next 6-number line.

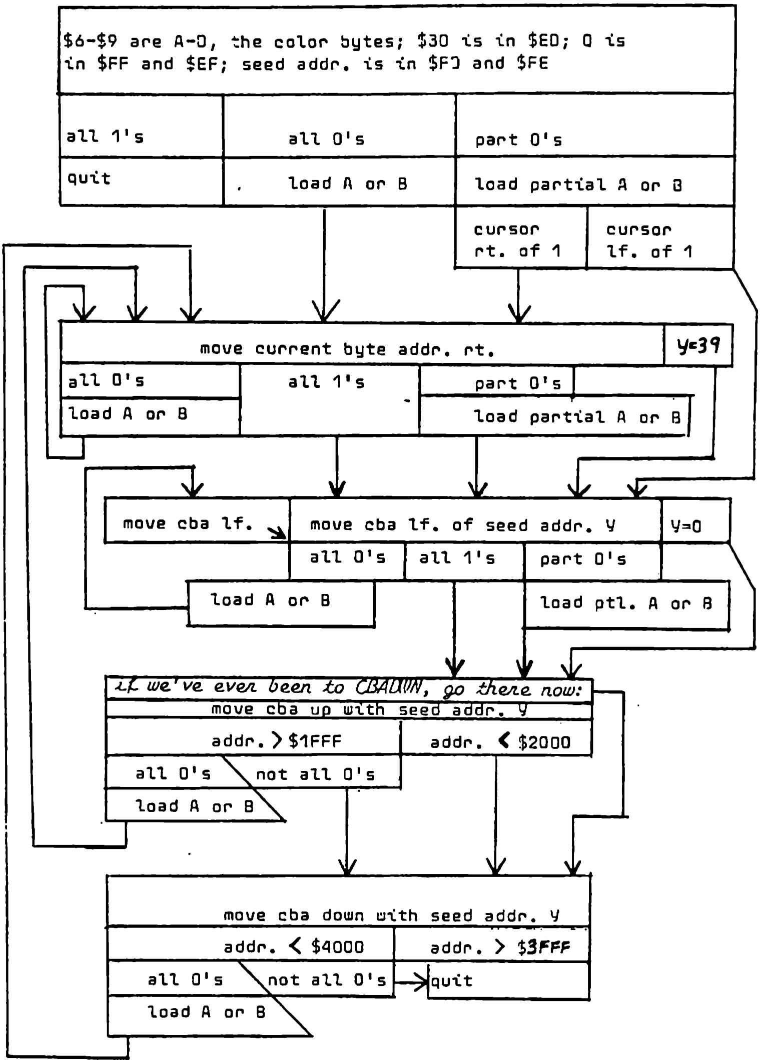
Using option 9 (part 1 or 2) in 4 or 8 of 28D, and 2 of 28A, you can create original pictures and/or shapes to color. Part 2 of option 9 of 4 or 8 of 28D will let you add shapes in any color to existing pictures, which may be re-SAVED.

On the next page is my FILL flowchart. I put color bytes A-D into $6-$9, the internal cursor's bit table byte from $30 into $ED, Q (hor. byte coord. #) into $FF and $EF, and the hi and lo of the vert. seed address into $FE and $FD respectively.

Then I check to see if the seed byte is all 1's, part 0's or all 0's. On all 1's we quit; on all 0's we load A or B (depending upon whether the seed address is odd or even --- $6 and $8 are for even addresses only and $7 and $9 are for odd only), and on part 0's we load part of a color byte into the screen byte. My routine calculates how much of the color byte goes into the screen byte.

If part of the byte was 0's, then we also calculate which side of the first 1 bit our cursor was on --- this tells us whether to begin moving left or right after the part-fill.

PRINTOUT #60


$6-$9 are A-D, the color bytes; $30 is in $ED; 0 is in $FF and $EF; seed addr. is in $FD and $FE
all 1's
all 0's
part 0's
quit
load A or B
load partial A or B
cursor rt. of 1
cursor lf. of 1
move current byte addr. rt.
y=39
all 0's
all 1's
part 0's
load A or B
load partial A or B
move cba lf.
move cba lf. of seed addr. y
y=0
all 0's
all 1's
part 0's
load A or B
load ptl. A or B
if we've ever been to CBADWN, go there now:
move cba up with seed addr. y
addr. > $1FFF
addr. < $2000
all 0's
not all 0's
load A or B
move cba down with seed addr. y
addr. < $4000
addr. > $3FFF
all 0's
not all 0's
quit
load A or B

Next we move the current byte address to seed address plus one, or right-wards, in simpler terms. If we're already at byte #39 for that line, we go down to the leftwards moving routines, since the first thing the **move cba right** routine does is increment to the next higher byte. If the byte is all Ø's, we load A or B, depending upon odd or even address, and then go looping back to do the move right routine **again** (The hi color bit is not counted when we're looking for "all Ø's.)

If the byte was all ones we go to **move cba left of seed addr.** If the byte was mixed we load partial A or B and then go to **move cba** (current byte addr.) **left of seed addr. Y** (which was stored in $FF).

In **move cba left of seed addr. Y** if Y is Ø, we go on to **move cba up** since our first move in this routine will be to decrement Y. If the byte is all Ø's we load A or B, depending upon odd or even addr., and then go to **move cba left**. This latter routine is the same as the one next to it on the chart except we don't use the seed addr. Y at all but continue moving left across the screen.

If the byte's all 1's we go to **move cba up**. (If the color bit is Ø it still counts as all 1's in any of these routines.) If the byte's part Ø's we load part of A or B and go to **move cba up**.

In **move cba up** we always use the same Y or hor. screen byte coord. as our seed address had. That's why the routine only goes upward until the first non-Ø byte which has no room for color is found, and the same can be said for moving down. If the new address is less than $2ØØØ we've got to move cba down because we're above the screen. If the byte is all Ø's we load A or B **if** we loaded in C and D on the line below, or we load C or B **if** we loaded A and B on the line below, and we leave these 2 new color bytes in our temporary color byte addresses ($1E and $1F). If the byte's not all Ø's then we go to **move cba down**.

If the byte's all Ø then our next move after filling it is to go back up to **move cba right**.

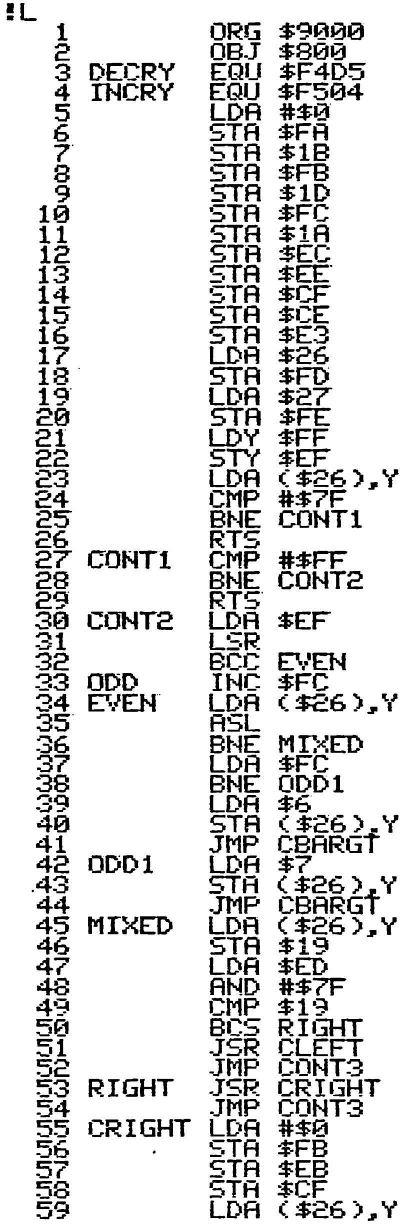
In **move cba down** we use the seed addr. Y again (the Y is the hor. byte coord. at which we began filling, if it's specifically the **seed addr.** Y) for hor. byte coord. If the addr. is greater than $3FFF or if the byte is not all Ø's we quit. If it is all Ø's we load in A or B if our last line above was C and D, or C and D if our last line above got A and B. If this is our first time coming to **move cba down**, we use C($8) and D($9) for sure. After A or B, or C or D, are loaded we go all the way back up to **move cba right**, which means now that we're on a new line we fill right as far as possible, then left as far as possible, but we skip CBAUP and come back to **move cba down**. Once we've done CBADWN we never use CBAUP again.

Always after getting to either **move cba up** or **move cba down** and finding a Ø's byte, we fill both ways (right, then left) on the line before doing any more vertical traveling.

You can see why you need to hold down the button on PDL#Ø while turning vert. coord. paddle knob #1 when you're in a small space: you're hunting for open pixels.

Now that you get the idea of how it works, let's look at the FILL source file, which assembles the binary FILL1:

PRINTOUT #61



```
!L
            ORG  $9000
            OBJ  $800
DECRY       EQU  $F4D5
INCRY       EQU  $F504
            LDA  #$0
            STA  $FA
            STA  $1B
            STA  $FB
            STA  $1D
            STA  $FC
            STA  $1A
            STA  $EC
            STA  $EE
            STA  $CF
            STA  $CE
            STA  $E3
            LDA  $26
            STA  $FD
            LDA  $27
            STA  $FE
            LDY  $FF
            STY  $EF
            LDA  ($26),Y
            CMP  #$7F
            BNE  CONT1
            RTS
CONT1       CMP  #$FF
            BNE  CONT2
            RTS
CONT2       LDA  $EF
            LSR
            BCC  EVEN
ODD         INC  $FC
EVEN        LDA  ($26),Y
            ASL
            BNE  MIXED
            LDA  $FC
            BNE  ODD1
            LDA  $6
            STA  ($26),Y
            JMP  CBARGT
ODD1        LDA  $7
            STA  ($26),Y
            JMP  CBARGT
MIXED       LDA  ($26),Y
            STA  $19
            LDA  $ED
            AND  #$7F
            CMP  $19
            BCS  RIGHT
            JSR  CLEFT
            JMP  CONT3
RIGHT       JSR  CRIGHT
            JMP  CONT3
CRIGHT      LDA  #$0
            STA  $FB
            STA  $EB
            STA  $CF
            LDA  ($26),Y
```

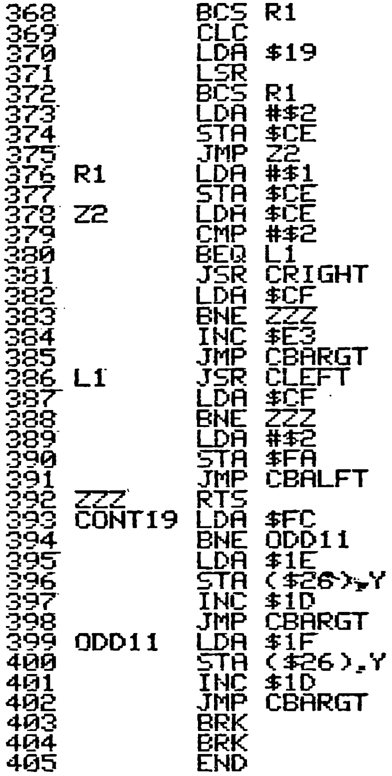
```
 60          ASL
 61 LOOP     INC $EB
 62          ASL
 63          BCC LOOP
 64          LDA $EB
 65          CMP #$2
 66          BCS SMLSFT
 67          INC $CF
 68          RTS
 69 SMLSFT   LDA $FC
 70          BNE ODD2
 71 EVEN1    LDA $1E
 72          JMP CONT4
 73 ODD2     LDA $1F
 74 CONT4    STA $19
 75          DEC $EB
 76          BEQ RTN
 77          LDA #$7
 78          SEC
 79          SBC $EB
 80          STA $EB
 81          STA $EC
 82          LDA $19
 83          CMP #$80
 84          BCC LOOP1
 85          INC $FB
 86 LOOPI    LSR
 87          DEC $EB
 88          BNE LOOP1
 89 LOOP2    ASL
 90          DEC $EC
 91          BNE LOOP2
 92          STA $19
 93          LDA ($26),Y
 94          ORA $19
 95          LDX $FB
 96          BEQ GO1
 97          ORA #$80
 98 GO1      STA ($26),Y
 99 RTN      INC $1A
100          RTS
101 CLEFT    LDA #$0
102          STA $FB
103          STA $EB
104          STA $CF
105          LDA ($26),Y
106 LOOP3    INC $EB
107          LSR
108          BCC LOOP3
109          LDA $EB
110          CMP #$2
111          BCS SMLSHT
112          INC $CF
113          RTS
114 SMLSHT   LDA $FC
115          BNE ODD3
116 EVEN2    LDA $1E
117          JMP CONT5
118 ODD3     LDA $1F
119 CONT5    STA $19
120          DEC $EB
121          BEQ RTN1
```

```
        LDA #$7
        SEC
        SBC $EB
        STA $EB
        STA $EC
        LDA $19
        CMP #$80
        BCC LOOP4
        INC $FB
LOOP4   ASL
        DEC $EB
        BNE LOOP4
LOOP5   LSR
        DEC $EC
        BNE LOOP5
        STA $19
        LDA ($26),Y
        ORA $19
        LDX $FB
        BEQ GO2
        ORA #$80
GO2     STA ($26),Y
RTN1    RTS
CONT3   LDA $1A
        BNE CBARGT
LFT     LDA #$0
        STA $FA
        STA $1A
        JMP CBALFT
CBARGT  LDA #$0
        STA $1A
        STA $FA
        CPY #$27
        BEQ LFT
        INY
        INC $EF
        LDA ($26),Y
        CMP #$7F
        BNE CONT6
        BEQ LFT
CONT6   CMP #$FF
        BNE CONT7
        BEQ LFT
CONT7   LDA #$0
        STA $FC
        LDA $EF
        LSR
        BCC EVEN3
ODD4    INC $FC
EVEN3   LDA ($26),Y
        ASL
        BNE MIXED1
        LDA $FC
        BNE ODD5
        LDA $1E
        STA ($26),Y
        JMP CBARGT
ODD5    LDA $1F
        STA ($26),Y
        JMP CBARGT
MIXED1  JSR CLEFT
CBALFT  LDX $E3
```

```
184        BEQ CBAL
185        LDA #$0
186        STA $E3
187        JMP CBAUP
188 CBAL   CPY #$0
189        BEQ CBAUP
190        INC $FA
191        LDA $FA
192        CMP #$2
193        BCS CONT8
194        LDY $FF
195 CONT8  DEY
196        STY $EF
197        LDA ($26),Y
198        CMP #$7F
199        BNE CONT9
200        BEQ CBAUP
201 CONT9  CMP #$FF
202        BNE CONT10
203        BEQ CBAUP
204 CONT10 LDA #$0
205        STA $FC
206        LDA $EF
207        LSR
208        BCC EVEN4
209 ODD6   INC $FC
210 EVEN4  LDA ($26),Y
211        ASL
212        BNE MIXED2
213        LDA $FC
214        BNE ODD7
215        LDA $1E
216        STA ($26),Y
217        JMP CBALFT
218 ODD7   LDA $1F
219        STA ($26),Y
220        JMP CBALFT
221 MIXED2 JSR CRIGHT
222 CBAUP  LDA $1D
223        BEQ Z
224        JMP CBADWN
225 Z      LDA $1E
226        CMP $6
227        BNE SIXSEV
228        LDA $1F
229        CMP $7
230        BNE SIXSEV
231        LDA $8
232        STA $1E
233        LDA $9
234        STA $1F
235        JMP CONT14
236 SIXSEV LDA $6
237        STA $1E
238        LDA $7
239        STA $1F
240 CONT14 JSR DECRY
241        LDY $FF
242        STY $EF
243        LDA $27
244        CMP #$3F
245        BCC CONT11
```

```
246         LDA  $26
247         CMP  #$D0
248         BCC  CONT11
249         JSR  INCRY
250         JMP  CBADWN
251 CONT11  LDA  ($26),Y
252         CMP  #$7F
253         BNE  CONT12
254         BEQ  CBADWN
255 CONT12  CMP  #$FF
256         BNE  CONT13
257         BEQ  CBADWN
258 CONT13  LDA  #$0
259         STA  $FC
260         TYA
261         LSR
262         BCC  EVEN5
263 ODD8    INC  $FC
264 EVEN5   LDA  ($26),Y
265         ASL
266         BEQ  Z0
267         LDA  $EE
268         BNE  Z1
269         LDA  ($26),Y
270         STA  $19
271         LDA  $ED
272         AND  #$7F
273         CMP  $19
274         BCS  R0
275         CLC
276         LDA  $19
277         LSR
278         BCS  R0
279         LDA  #$2
280         STA  $EE
281         JMP  Z1
282 R0      LDA  #$1
283         STA  $EE
284 Z1      LDA  $EE
285         CMP  #$2
286         BEQ  L0
287         JSR  CRIGHT
288         LDA  $CF
289         BNE  CBADWN
290         INC  $E3
291         JMP  CBARGT
292 L0      JSR  CLEFT
293         LDA  $CF
294         BNE  CBADWN
295         LDA  #$2
296         STA  $FA
297         JMP  CBALFT
298 Z0      LDA  $FC
299         BNE  ODD9
300         LDA  $1E
301         STA  ($26),Y
302         JMP  CBARGT
303 ODD9    LDA  $1F
304         STA  ($26),Y
305         JMP  CBARGT
306 CBADWN  INC  $18
```

```
307          LDA $1B
308          CMP #$2
309          BCS BENHRE
310          LDA $8
311          STA $1E
312          LDA $9
313          STA $1F
314          LDA $FD
315          STA $26
316          LDA $FE
317          STA $27
318          JMP CONT15
319 BENHRE   LDA $1E
320          CMP $6
321          BNE SIXSV1
322          LDA $1F
323          CMP $7
324          BNE SIXSV1
325          LDA $8
326          STA $1E
327          LDA $9
328          STA $1F
329          JMP CONT15
330 SIXSV1   LDA $6
331          STA $1E
332          LDA $7
333          STA $1F
334 CONT15   JSR INCRY
335          LDY $FF
336          STY $EF
337          LDA $27
338          CMP #$21
339          BCS CONT16
340          LDA $26
341          CMP #$28
342          BCS CONT16
343 RTT      RTS
344 CONT16   LDA ($26),Y
345          CMP #$7F
346          BNE CONT17
347          BEQ RTT
348 CONT17   CMP #$FF
349          BNE CONT18
350          BEQ RTT
351 CONT18   LDA #$0
352          STA $FC
353          TYA
354          LSR
355          BCC EVEN6
356 ODD10    INC $FC
357 EVEN6    LDA ($26),Y
358          ASL
359          BEQ CONT19
360          INC $1D
361          LDA $CE
362          BNE Z2
363          LDA ($26),Y
364          STA $19
365          LDA $ED
366          AND #$7F
367          CMP $19
```



```
368          BCS R1
369          CLC
370          LDA $19
371          LSR
372          BCS R1
373          LDA #$2
374          STA $CE
375          JMP Z2
376 R1       LDA #$1
377          STA $CE
378 Z2       LDA $CE
379          CMP #$2
380          BEQ L1
381          JSR CRIGHT
382          LDA $CF
383          BNE ZZZ
384          INC $E3
385          JMP CBARGT
386 L1       JSR CLEFT
387          LDA $CF
388          BNE ZZZ
389          LDA #$2
390          STA $FA
391          JMP CBALFT
392 ZZZ      RTS
393 CONT19   LDA $FC
394          BNE ODD11
395          LDA $1E
396          STA ($26),Y
397          INC $1D
398          JMP CBARGT
399 ODD11    LDA $1F
400          STA ($26),Y
401          INC $1D
402          JMP CBARGT
403          BRK
404          BRK
405          END
```

Line 1 and 2 say assemble it at $800 but have it work right at $9000, so you'll have to save it at $800 while in LISA but BLOAD it at $9000 later and save it **there** for use.

Line 3-16 give Applesoft DECRY and INCRY addresses because you'll need to increment/decrement vert. addr.; also, flags get zeroed.

Line 17-20 record base addr. in permanent locations so we may use it when we need to find "one line below where we started" in the CBADWN routine at line 306. The "base address" is the address of the screen coordinate minus the horizontal displacement which gets done by indexing. The hor. displacement is also recorded --- in $EF and $FF; we'll monitor our displacements with $EF, but leave $FF as an important # to remember.

Line 23-29 loads current byte and if it's all white (white1 would be $7F and white2 would be $FF) we quit.

Line 30-33 we see if our hor. displacement is even or odd. This will tell us if our addr. is even or odd, since **all** base addresses stored in $26 and $27 are **even** (since each line goes up 1024,40, or 128 over the previous one).

The way we tell odd is by shifting the 0 bit into the carry. Since **all** bytes with 0 in their 0 bits are even, we need only find out the 0 bit status to know "odd or even" for the entire byte (the bit values are 128,64,32,16,8,4,2, and 1, so the 0 bit, with 1 value, is **oddness** bit).

Line 34-36 now we load and shift (left) the current byte to discard the color bit (because a 0 byte or an $80 byte would be equally black and invisible). If the byte that's left isn't 0, we have a mixed byte, which means containing both 1's and 0's, so we go to the MIXED routine, which deals with loading **part** of the color byte (can be $6-$9) into the 0 bit part of the current hi-res byte.

Line 37-38 now we check out the odd/even flag ($FC) which we just got through conditioning, which means setting to 0 or 1 according to conditions. If it's set, we go to line 42 because we're at **odd** addr.

Line 39-41 since the address is even, we load the even ($6) color byte in and store it on the screen and then jump all the way to CBARGT at line 151. This is the **current byte address moving right** routine ---consult flowchart. Remember, our screen byte was 0 or we'd have gone to MIXED by now. As long as bytes are 0, we just keep sailing along in the direction we're going (we start with right). Only when an edge-byte (not all 0's) is encountered do we need to go to special routines or fill in another direction.

Line 42-44 loads the odd color byte onto the screen byte address and then jumps to CBARGT.

Line 45-46 stores screen byte temporarily in $19.

Line 47-50 loads last internal cursor bit position byte, which was loaded into 237 ($ED) in the BASIC driver program Palette. ANDing with $7F zeroes hi bit which will get in the way of comparisons coming up next. CMP$19 compares your cursor's position (the screen dot at the seed-point) with the hi-res byte. If the cursor is greater than the screen byte then it's to the (visible) right of the screen byte 1's, and we're sent to RIGHT which sends us to CRIGHT, which means "cursor's to the right of any hi-res on bits so let's move right next." Cursor being "greater than" **on** screen byte's means it's in a bit of greater value; and visible bits on the screen have these values:

binary bits:	1	1	1	1	1	1	1
values:	1	2	4	8	16	32	64
bit #:	0	1	2	3	4	5	6

Line 51 if the cursor wasn't right of the hi-res on bits it was left, so we'll go there next. CLEFT means "cursor's left of on bytes, move left next."

Line 52-54 once either CRIGHT or CLEFT are finished we'll be going to CONT3 where we'll figure out where to move current byte address, and whether we're filling to the left or right next.

Line 55-58 zero the "hi bit set" and shift counter flags ($FB,$EB) and then zero the "no room for color" flag ($CF).

Line 59-63 load screen byte, shift out hi bit, then shift until an ON bit is found, using $EB as shift counter and branch if carry clear (BCC) to determine that no ON bits have been discovered by getting shifted into the carry bit.

Line 64-68 we load the shift counter into the accumulator and compare it to 2. If it's 2 or better we continue with CRIGHT since the required 2 open bits are present. We set the "no room for color" flag in line 67 if there was less than 2 shifts recorded on the counter, and then quit the routine --- there's no room for part of any color byte to be added.

Line 69-72 load and check even/odd flag and load in color byte $6 or $8 if even and $7 or $9 if odd. Odd/even was determined, for the seed byte, at line 26-29. $6 will go in even seed byte, but when we've moved up or down a few lines we'll use $8 sometimes for even and $9 sometimes for odd. The temporary color bytes $1E and $1F hold either $6 and $7 or $8 and $9.

Line 73-76 here we store color byte temporarily in $19 and then decrement counter and check if it's down to Ø --- if it is we exit routine, because we need at least 2 Ø's before we'll add color to the byte.

Line 77-81 we subtract counter from 7 and store the result in counter address ($EB) and in countersaver address $EC. This new # represents the shifts required before we can put color byte and screen byte together. Remember, we're shifting the color byte from perhaps Ø1Ø1Ø1Ø1 to perhaps Ø1Ø1ØØØØ and then back again to the original positions minus the excess 1's, so we end up with ØØØØØ1Ø1 for color bytes (the hi-res byte would have been ØØ11ØØØØ or ØØØ1ØØØØ). The nature of ASL and LSR is that the 1's shifted out are gone for good, so a 7-dot-wide color byte can soon turn into a 2-6-dot wide color byte.

Line 82-85 here we see if color byte uses ON color bit. If it does we set the "hi bit set" flag ($FB).

Line 86-88 here the first half of the shifting of the color byte occurs, exhausting counter $EB.

Line 89-91 we shift the color byte bits back like it was, only MINUS the unwanted ON bits. We use up counter $EC in the process.

Line 92-94 puts the "fixed" color byte into $19 and puts the screen byte in the accumulator and ORAs the 2 operands. This adds them up --- a one in either operand gets a one in the accumulator-stored results.

Line 95-97 if the color byte's hi bit was on before we monkeyed with it, we turn it back on.

Line 98-1ØØ accumulator is put into screen memory with its screen byte/color byte combination and $1A is incremented to tell us, once we get to CONT3, that we've just come back from CRIGHT so send me next to CBARGT --- we're moving right. We then exit the routine and find ourselves in CONT3.

Line 1Ø1-144 is the same as 55-1ØØ with the following exceptions:

a) we're calling it CLEFT because we were sent here if the cursor is **left** of the ON screen bits.

b) it doesn't set $1A flag, since that's only to tell us that we've returned from CRIGHT.

c) the color byte we end up with after shifting is done is on the **left** (visible) side of the byte, and all shifts are done in opposite order (by interchanging LSR and ASL).

Line 145-146 here we are at CONT3 where either CRIGHT or CLEFT end up **if we're still on the seed byte.** If the "back from CRIGHT" flag is up and waving we go to CBARGT.

Line 147-150 If that flag's off, we 0 the "left" flag ($FA) and go to CBALFT. This will be explained later. See flow-chart for a hint.

Line 151-155 we're in CBARGT; we zero the "left" flag and see if we're at 39; if so we jump to CBALFT because our first endeavor in CBARGT will be incrementing our hor. displacement byte $EF, and only up to 39 is allowed.

Line 156-164 we increment Y and $EF (the former is our indexing register, which uses hor. displacement values sometimes pulled from $EF), then load the screen byte and find out if it's all 1's (discounting high bit again). If it is we go to CBALFT.

Line 165-170 we zero the odd/even flag ($FC) and load the hor. displacement and use our shift-and-test-carry test for odd/evenness, and set the flag if it's odd.

Line 171-173 now we test the screen byte to see if it's 0, after discarding the hi bit by use of ASL. We go to MIXED1 if it's not 0, which sends us to CLEFT --- we can go no further right.

Line 174-182 test for odd/even and load color bytes accordingly, as always and then jump back to CBARGT --- since we're only here if the screen byte was 0, there's no ON-bits yet found (to indicate shape edge) which would give us reason to cease filling to the right.

Line 183-221 is CBALFT, which is like CBARGT, except:

a) we fill and move leftwards, not rightwards.

b) we test to see if we've **just done** CBALFT; if so, we decrement the hor. displacement by 1 and continue filling. The way we see if we've **just done** CBALFT is by checking the "left" flag ($FA). If it's greater than 1 we **just did** this routine. Other routines zero this flag before sending us here, but CBALFT doesn't so the INC $FA in **Line 190** will add up to 2 or more if we've "been here before." **Lines 191-193** catches "2 or greater" for this flag and sends us to **line 195 without loading Y with $FF first.** The importance of this is that if we always loaded $FF into Y we'd never move more than 1 byte left. Remember that $FF stores the seed addresses hor. displacement permanently, so decrementing $FF gives us just what we want for jumping from "done with CBARGT due to non-0 byte" to "going to $FF minus one to start filling left." However, as the flowchart shows, once we're doing 2 or more CBALFT routines in a row, we must simply DEY **without** loading $FF, so we can move as far left on the screen as necessary without problem.

c) we also test to see if we're at hor. byte 0 in line 188-189 and get sent to CBAUP if we're at 0 --- see flowchart.

d) if we're at a mixed byte we get sent to partially fill the current address with the color byte at routine CRIGHT (we can go no farther leftwards) after which we're sent to CBAUP. This differs from our CBARGT routine which sends us to CBALFT afterwards --- keep following along with the flow chart and all will remain clear.

e) at the beginning of CBALFT we check flag $E3, the flag that stops you from doing CBALFT **after** CBARGT if you've just come from line 291 or 385 --- from these places we want only to fill rightwards, not leftwards.

Line 222-230 this is CBAUP, and we need to remember that color bytes of $6 and $7 **or** $8 and $9 are unchanged as long as you make only horizontal moves. But this is a vertical (up) mover, so we load **current color byte** addresses $1E and then $1F and see if they're $6 and $7; if not, we go to **line 236.** (We also checked the "already doing CBADWN flag which is $1D. See flow chart. Once we've hit an obstacle in CBAUP, we go to CBADWIN, and although we continue we to go through CBALFT and CBARGT, we never again touch CBAUP, since we now **know** there's a "non-0-byte-barrier" so it's no use. We go right, left, up, right, left, up until obstacle, then we go right, left, down, right, left down until quit --- there is **no** return to CBAUP --- which is the message of $1D.)

Line 231-235 if our latest color bytes were $6 and $7 we now put $8 and $9 in their place --- every other horizontal line going up gets $8 and $9 --- the rest get $6 and $7.

Line 236-239 if we **were** using $8 and $9, we now switch our temporary color bytes ($1E and $1F) to $6 and $7.

Line 240-242 we move our base (not seed) address (which means which vertical line address is the 0 byte of the line at) up a line (decrement Y coord. which has nothing whatever to do with the Y register), load in see addr. hor. byte displacement, store that in temporary hor. displacement.

Line 243-250 we check to see if we've decremented so high it wrapped around to the bottom of the screen (like when vector shapes' tops are on the bottom of the screen because they were drawn too high); if it has, we increment and go to CBADWN.

Line 251-257 we check to see if the byte is all ones --- if it is we go to CBADWN.

Line 258-263 we zero odd/even flag, do an LSR and check for odd/evenness and set ($FC) odd/even flag accordingly.

Line 264-266 load byte, shift out 7 bit, and go to line 267 if it's a non-zero byte. Remember, going horizontally we do partial color byte filling but going vertically we continue only as long as part-0 fillable bytes continue above or below seed byte.

Line 267-268 checks the $EE flag, which does 2 things: tells us if we've been in CBAUP before when we had partial 0's at our CBA (current byte addr.), and it also specifies whether the cursor is left or right of the ON bits in the CBA (2 means left, 1 mean right).

Line 269-283 loads CBA byte, stores it in $19, loads bit table byte of cursor, zeros high bit, compares cursor's position with ON bits in CBA byte, if cursor's left of ON bit, flag $EE gets a 2, if right, $EE gets a 1.

Line 284-286 checks flag --- if it says "left" then we're sent to line 292, otherwise if it's "right" then we continue at line 287.

Line 287-297 we're sent to part-fill CBA with color byte, then we check "no room for color" flag ($CF) --- if it's set we're at a byte with no room for adding color pixel(s) so we abandon CBAUP for good and head for CBADWN. If we used CRIGHT we ignore $FA flag, but if we just returned from CLEFT and the $CF flag is OFF, then we head for CBALFT after putting a 2 into flag $FA, the flag that says we've just been to CBALFT and need to do it again without going to the # in $FF minus one. Our procedure is merely a double-safeguard to make sure we merely move leftwards one byte.

In lines 29Ø-291 we set the $E3 flag since we don't want to go to CBALFT after going to CBARGT --- we want to fill rightwards only, since in line 287 we went for a CRIGHT partial-fill. The shape boundary is in that CRIGHT byte, so moving hor. displacement left in CBALFT would start filling out of the section we were filling.

Line 298-3Ø5 byte is Ø, load color byte according to odd/even flag ($FC) and then go right to fill as far as possible (followed by left, up, right, left, etc.)

Line 3Ø6-3Ø8 check the "been down before" flag --- if it's 2 or more then go to BENHRE where you'll simply insert into the temporary color byte locations the opposite bytes than were there; $6 goes to $8 and $7 goes to $9 or vice versa. You see, once you start doing downward filling, there's nothing to do but swtich back and forth with these color bytes ($6 and $8 are for even hor. displ. and $7 and $9 are for odd hor. displ.). But the very first time you enter this subroutine, the $1B flag ("been down before") will get incremented to 1 only, so you'll get $8 and $9 as color bytes and $FD as $26 and $FE as $27. The $8 and $9 are because you're **one line** under the seed byte which for sure started with bytes $6 and $7, and the only way you can get the correct color is to have all 4 bytes in proper alternating relationship to each other and at proper odd/even addresses. The $26 and $27 are the base address bytes affected by INCRY and DECRY or by loading in the original vertical base address of the seed address stored in $FD and $FE in **lines 18 and 2Ø.**

Line 3Ø9-318 you haven't been here, so in go $8 and $9 and $FD and $FE, then we jump to **line 28.**

Line 319-333 if we're here, we **have** been here before so we put in the opposite color bytes from those presently there.

Line 334-336 we increment the base address so we are one line lower, then we get the seed address' hor. displ. and stick it into the temporary hor. displ. location ($EF).

Line 337-343 we check to see if we've gone so low we're wrapped around to the top, in which case we'd have a base address whose hi byte would be $2Ø and whose low byte would be Ø. We quit if there's wrap-around evidence.

Line 344-356 first we check for all ones. We quit if byte is all ones. Then we zero the odd/even flag ($FC) and check for odd/even by LSR and BCC again. We set the odd/even flag accordingly.

Line 357-359 we load the screen byte and if it's Ø we go to CONT19.

Line 36Ø-392 same as line 267-297 except INC $1D in line 336 is the setting of the "once we've been **down,** never go back **up**" flag. Also, if the "no room for color" flag ($CF) is found to be set, the whole fill routine terminates, rather than being sent to CBADWN, which is what happens in lines 289 and 294 of CBAUP. The comments for 29Ø-291 apply also to 384-385.

Line 393-4Ø2 load in the appropriate color byte, if the screen byte is Ø, then jump back to CBARGT to fill right, left, and then down again until a non-Ø byte is hit. We also increment the "been down before" flag ($1D) so the **next** time we're in CBADWN we'll simply load the opposite 2 color bytes rather than setting the color bytes to $8 and $9, thinking we're only one line beneath the seed byte. Either wrap-around or non-0 screen byte which has no room for color will get us to exit filling, in this subroutine.

I hope you have a great time using this routine in Palette or your own programs. I never went to kindergarten so I'm making up for lost time on this one --- coloring is fun! Don't be afraid to complement your coloring and painting endeavors in palette with the filtering, complementing, line-fixing, color-bit setting, and color-bit zeroing options in 8 of 28D. Some really wild combinations are possible.

Also, learn as soon as you can how to do what happens in 9 of 28D, the Color Fill Demo (only do it better! --- mistakes are what 9 of 28D is meant to illustrate --- as well as methodology).

Once you get an "instant scene" running smoothly, it will act like, and probably be constructed like, A of 28D, the Good Color Fill Demo. If you haven't yet run that, do it now. It feels nice to know you can slap together various shapes and load them into a quick composite picture that you can seed-fill automatically in a couple seconds with scores of nice colors and patterns.

30 COLOR FILTERING

In 8 of 28D you'll discover several interesting options besides 9, where you move vector or block-shapes around and press button #1 to copy the shape or #0 to exit. Now load in COMPOSITE 2.

Run 8 of 28 now and choose option 17 --- fix all white lines. Watch the vertical lines widen. Now choose option 9 and go for choice 2 ---vector shape. Load in POS which is a stretching exercise table from our Physical Life Dynamic disk. Ask for shape 1-14 (just one of these) and position him in the doorway and hit button #1 and then move away ---the shape should be there. Now exit with button #0.

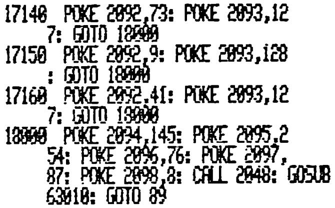
Run option 14 and the screen will inverse. Then run option 13 to filter out blue. The filtering blue or orange affects only bytes with the high bit on, which isn't many. Now do it again with violet. Things get green since "white one" is green **and** violet and both of these are hi-bit-OFF colors. Now run 16 to zero entire screen's color bit. The few places that may have been splotchy from having hi bits on as a result of complementing and white line fixing have now joined the rest of the screen and turned green. Now run 15 to turn all color bits on the screen to 1. If you'll notice in the color bit table in the hi-res color chapter, the only difference between green and orange is a set high bit for orange. Your screen is now orange if it's tuned correctly. A part of your yoga person's body may have dropped out by now, which lets you see why the fix white line option could have considerable value.

Run option 14 to complement the screen --- black lines on an orange background will change to white lines on a blue background. Check the color byte tables again and you'll see that if you shift over orange one bit you'll have blue. The complement (opposite of a black (00) pixel is a white (11) one and vice versa, and the complement of a blue (01) pixel is an orange (10) one, with hi bit set for each. Now run #17 and notice that if every other bit on the screen is on and then you double the width of every "1" part of the screen, the screen turns to all ones --- white.

The color "filter" is merely a mach. lang. masking program that masks out all the screen bits that make up whatever color you choose of the 4 colors green, violet, orange or blue. It sees white 1 as green and violet and white 2 as orange and blue. The former are all hi-bit-off colors, the latter are hi-bit-on colors. So if you run color filters on white, then the result is that half of what makes up the white will filter out. To leave white alone have it be opposite in hi-bit setting from the colors you filter. A little mach. lang. program called **filter** is the masker here.

But let's look at how I performed the hi bit setter, zeroer, and the complementer routines:

PRINTOUT #62



```
17140  POKE 2092,73: POKE 2093,12
    7: GOTO 18000
17150  POKE 2092,9: POKE 2093,128
    : GOTO 18000
17160  POKE 2092,41: POKE 2093,12
    7: GOTO 18000
18000  POKE 2094,145: POKE 2095,2
    54: POKE 2096,76: POKE 2097,
    87: POKE 2098,8: CALL 2048: GOSUB
    63010: GOTO 89
```

For complementing, see line 17140 above. 73 is 49 hex and 127 is 7F hex. If you look up the opcode and operand #s you get EOR #$7F. To complement a byte and leave the hi bit alone, we EOR with $7F (the # simply means immediate addressing so the **value** 127 is used rather than the contents of address $7F). **One** result when any 2 bits are different in the operands.

For setting all hi bits, we ORA #$80 which makes the accumulator's hi bit get set no matter what it now is, because ORA gets a 1 unless **both** bits compared are 0. Notice that line 17150 puts 9 (hex 9) and 128 (hex $80) into 2 sequential locations. Looking up 9 you'll find ORA.

In line 17160 you'll find 41 (hex 29) and 127 (hex 7F). $29 is AND so AND #$7F is the instruction. This zeros all hi bits because only if **both** operands' bits are 1 do you get a 1 in ANDing.

So how did I make all 8192 screen locations do the above things? (Actually 192 times 40 is only 7680 --- there are 64 sets of 8(=512) invisible screen locations --- study the reference manual's hi-res map.)

Well --- I cheated! I BLOADed good old WHITELINE 1 and then POKEd in the necessary changes and CALLed the program at 2048, its usual CALLing address. Now go back to Chapter 18 and check out WHITELINE, the source code. The lines from 24-45 would be of no use for a program that did complement, hi bit set, and hi bit zero. These are conditional lines that have to do with test and branch. The 3 functions we're interested in are unconditional complementing and bit zeroing or setting. So all we need after line 23 is to EOR, AND, or ORA, and then STA ($FE), Y and decrement address by use of SUB --- lines 46-54.

So the EOR #$7F of line 17140 POKES in at line 24 and then line 18000 POKES in STA ($FE), Y ($91 $FE, or 145 254) and then jumps to SUB which is at $861, and JMP SUB would then be $4C $61 $08 (76 97 8). So line 18000 CALLs the amended WHITELINE 1 routine once the POKEs are in and the hi-res addresses get operated on by Physician Fudge's Fixer.

If you ever want to determine # of bytes on screen that have their color bytes set, type NEW and then:

```
10 POKE−16304,0:POKE−16297,0
20 FOR Q=8192 TO 16383:P=PEEK(Q): IF P > 127 THEN
PP=PEEK(−16336);Z=Z+1;POKEQ,255
30 NEXT: TEXT:?"# BYTES:"Z
```

But perhaps you're curious as to the applications of all this hi-res color manipulation. It's "fairly obvious" --- but those are often famous last words.

1) You're not stuck with hi-res "paintings" as they are --- you can change colors to other colors, inverse the screen, and then filter one of white's colors out after inversing. The result will be your entire picture is filled with a color --- which would be a great effect as is, or would be convenient if you were going to do some 2 color printing and were going to photograph the screen image. You can manipulate colors with all these routines until you have somewhat the equivalent of color separations.

2) If you filled a whole picture this way you could hit 9 or Space Bar while in option 11 of 4 of 28D (Palette) and 0 for color and erase certain areas which you could re-color in Palette (remember that 8 of 28D does the filtering and complementing and 4 of 28D does the filling and painting --- both do the white line fix) --- this would be especially viable when you needed most of the screen one color.

3) Notice the eerie effects of going to Palette and loading in Composite 2 and fixing the white lines, then going to 8 of 28D (use Space Bar upon entry so you don't lost the picture), then setting the hi bit (notice the ½ bit shift of the picture --- hi bit colors really are positioned ½ bit over from low bit colors), and then filtering out orange. The blue screen is rather a formidable mood for a scene.

4) Do abstract art with the paint brushes and use **fill** in black holes or around what you do, and then use filtering/complementing/etc. for effect.

YTABLE USAGE ---
A NICE SPEED IMPROVEMENT

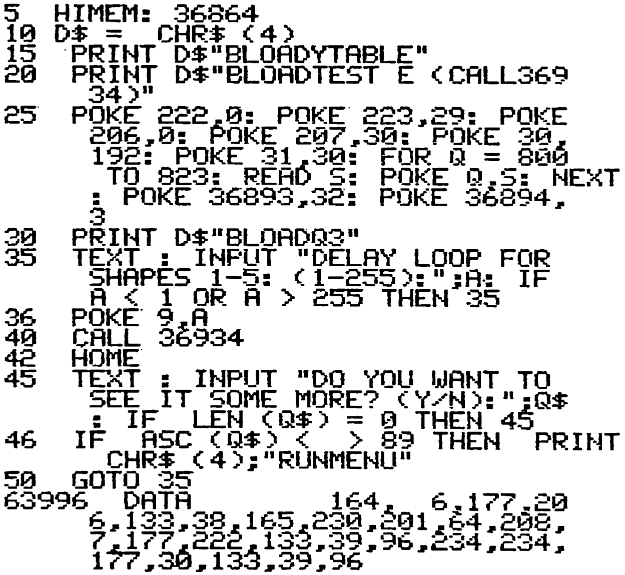
31

Run 4 of 28C

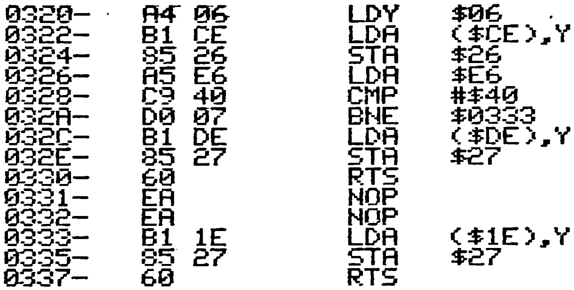
It takes about 2.75 seconds for the flying saucer to get across the screen, and it's moving only one bit per move. There are about 114 bytes that load in per shape-draw, and all have to be XDRAWN over for erasure. The shape moves about 240 times in its journey, or 87 times a second. This means that 22272 bytes get loaded from a block-shape table to screen addresses every second, which is 20% (at least) faster than the HPOSN routine can do. (There are animation schemes where HPOSN gets run much more often, so the advantage of YTABLE over HPOSN is increased much more.)

Here's the program that you find in 4 of 28C, and here's the mach. lang. YTABLE accesser that gets POKEd into the memory at $320 using the line 25 FOR-NEXT and line 63996 DATA statements, and here's the YTABLE list-out:

PRINTOUT #63

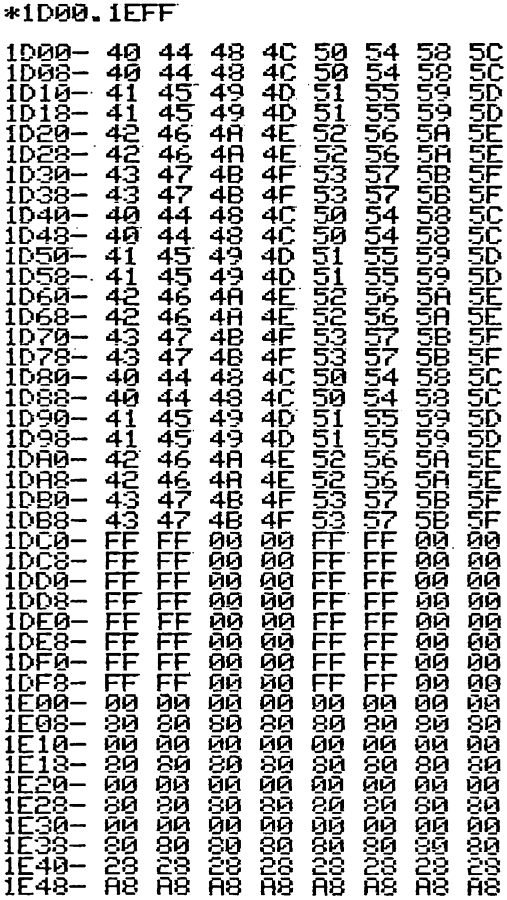


```
5   HIMEM: 36864
10  D$ =  CHR$ (4)
15   PRINT D$"BLOADYTABLE"
20   PRINT D$"BLOADTEST E (CALL369
      34)"
25   POKE 222,0: POKE 223,29: POKE
      206,0: POKE 207,30: POKE 30,
      192: POKE 31,30: FOR Q = 800
       TO 823: READ S: POKE Q,S: NEXT
      : POKE 36893,32: POKE 36894,
      3
30   PRINT D$"BLOADQ3"
35   TEXT : INPUT "DELAY LOOP FOR
      SHAPES 1-5: (1-255):";A: IF
      A < 1 OR A > 255 THEN 35
36   POKE 9,A
40   CALL 36934
42   HOME
45   TEXT : INPUT "DO YOU WANT TO
      SEE IT SOME MORE? (Y/N):";Q$
      : IF  LEN (Q$) = 0 THEN 45
46   IF  ASC (Q$) <  > 89 THEN  PRINT
       CHR$ (4);"RUNMENU"
50   GOTO 35
63996  DATA         164, 6,177,20
      6,133,38,165,230,201,64,208,
      7,177,222,133,39,96,234,234,
      177,30,133,39,96
```

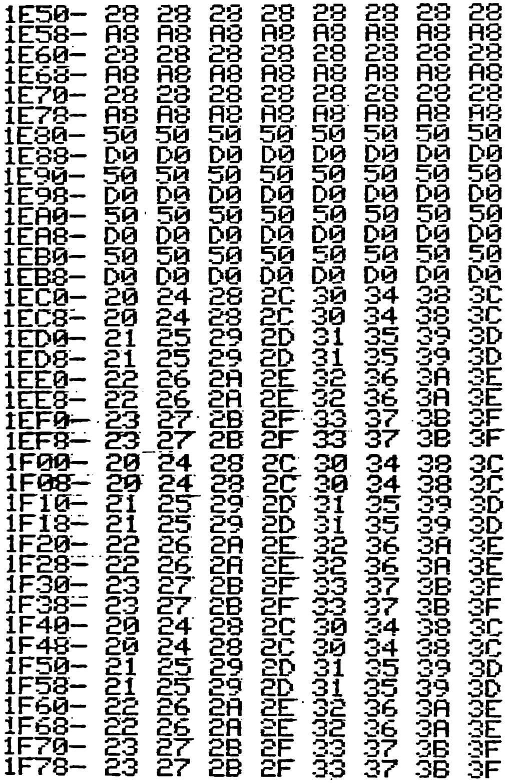


```
0320-   A4 06       LDY   $06
0322-   B1 CE       LDA   ($CE),Y
0324-   85 26       STA   $26
0326-   A5 E6       LDA   $E6
0328-   C9 40       CMP   #$40
032A-   D0 07       BNE   $0333
032C-   B1 DE       LDA   ($DE),Y
032E-   85 27       STA   $27
0330-   60          RTS
0331-   EA          NOP
0332-   EA          NOP
0333-   B1 1E       LDA   ($1E),Y
0335-   85 27       STA   $27
0337-   60          RTS
```

PRINTOUT #64



```
*1D00.1EFF

1D00- 40 44 48 4C 50 54 58 5C
1D08- 40 44 48 4C 50 54 58 5C
1D10- 41 45 49 4D 51 55 59 5D
1D18- 41 45 49 4D 51 55 59 5D
1D20- 42 46 4A 4E 52 56 5A 5E
1D28- 42 46 4A 4E 52 56 5A 5E
1D30- 43 47 4B 4F 53 57 5B 5F
1D38- 43 47 4B 4F 53 57 5B 5F
1D40- 40 44 48 4C 50 54 58 5C
1D48- 40 44 48 4C 50 54 58 5C
1D50- 41 45 49 4D 51 55 59 5D
1D58- 41 45 49 4D 51 55 59 5D
1D60- 42 46 4A 4E 52 56 5A 5E
1D68- 42 46 4A 4E 52 56 5A 5E
1D70- 43 47 4B 4F 53 57 5B 5F
1D78- 43 47 4B 4F 53 57 5B 5F
1D80- 40 44 48 4C 50 54 58 5C
1D88- 40 44 48 4C 50 54 58 5C
1D90- 41 45 49 4D 51 55 59 5D
1D98- 41 45 49 4D 51 55 59 5D
1DA0- 42 46 4A 4E 52 56 5A 5E
1DA8- 42 46 4A 4E 52 56 5A 5E
1DB0- 43 47 4B 4F 53 57 5B 5F
1DB8- 43 47 4B 4F 53 57 5B 5F
1DC0- FF FF 00 00 FF FF 00 00
1DC8- FF FF 00 00 FF FF 00 00
1DD0- FF FF 00 00 FF FF 00 00
1DD8- FF FF 00 00 FF FF 00 00
1DE0- FF FF 00 00 FF FF 00 00
1DE8- FF FF 00 00 FF FF 00 00
1DF0- FF FF 00 00 FF FF 00 00
1DF8- FF FF 00 00 FF FF 00 00
1E00- 00 00 00 00 00 00 00 00
1E08- 80 80 80 80 80 80 80 80
1E10- 00 00 00 00 00 00 00 00
1E18- 80 80 80 80 80 80 80 80
1E20- 00 00 00 00 00 00 00 00
1E28- 80 80 80 80 80 80 80 80
1E30- 00 00 00 00 00 00 00 00
1E38- 80 80 80 80 80 80 80 80
1E40- 28 28 28 28 28 28 28 28
1E48- A8 A8 A8 A8 A8 A8 A8 A8
```



```
1E5Ø-  28 28 28 28 28 28 28 28
1E58-  A8 A8 A8 A8 A8 A8 A8 A8
1E6Ø-  28 28 28 28 28 28 28 28
1E68-  A8 A8 A8 A8 A8 A8 A8 A8
1E7Ø-  28 28 28 28 28 28 28 28
1E78-  A8 A8 A8 A8 A8 A8 A8 A8
1E8Ø-  5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø
1E88-  DØ DØ DØ DØ DØ DØ DØ DØ
1E9Ø-  5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø
1E98-  DØ DØ DØ DØ DØ DØ DØ DØ
1EAØ-  5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø
1EA8-  DØ DØ DØ DØ DØ DØ DØ DØ
1EBØ-  5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø 5Ø
1EB8-  DØ DØ DØ DØ DØ DØ DØ DØ
1ECØ-  2Ø 24 28 2C 3Ø 34 38 3C
1EC8-  2Ø 24 28 2C 3Ø 34 38 3C
1EDØ-  21 25 29 2D 31 35 39 3D
1ED8-  21 25 29 2D 31 35 39 3D
1EEØ-  22 26 2A 2E 32 36 3A 3E
1EE8-  22 26 2A 2E 32 36 3A 3E
1EFØ-  23 27 2B 2F 33 37 3B 3F
1EF8-  23 27 2B 2F 33 37 3B 3F
1FØØ-  2Ø 24 28 2C 3Ø 34 38 3C
1FØ8-  2Ø 24 28 2C 3Ø 34 38 3C
1F1Ø-  21 25 29 2D 31 35 39 3D
1F18-  21 25 29 2D 31 35 39 3D
1F2Ø-  22 26 2A 2E 32 36 3A 3E
1F28-  22 26 2A 2E 32 36 3A 3E
1F3Ø-  23 27 2B 2F 33 37 3B 3F
1F38-  23 27 2B 2F 33 37 3B 3F
1F4Ø-  2Ø 24 28 2C 3Ø 34 38 3C
1F48-  2Ø 24 28 2C 3Ø 34 38 3C
1F5Ø-  21 25 29 2D 31 35 39 3D
1F58-  21 25 29 2D 31 35 39 3D
1F6Ø-  22 26 2A 2E 32 36 3A 3E
1F68-  22 26 2A 2E 32 36 3A 3E
1F7Ø-  23 27 2B 2F 33 37 3B 3F
1F78-  23 27 2B 2F 33 37 3B 3F
```

In case you haven't gleaned this already from earlier chapters, YTABLE is a method of avoiding base addr. calculations. Go back to chapter 22 and check out the difference between a scroller (TEST47) that uses base calculations and one (UPSCRL) that uses YTABLE.

In the routine above at $32Ø, we load the temporary vertical coord. ($6) into Y and index LDA ($CE), Y with this value and then store it in $26, where we're storing screen addr. lo byte. Also we check in $326 to see if we're drawing on screen 1 or 2 ($E6 is always $2Ø if we're on screen 1 and $4Ø if we're on screen 2). If we're on one, we get our hi byte at the table that starts at $1ECØ, an address which is stored in $1E and $1F. You can see how it got loaded there --- see line 25 where we POKE 3Ø,192 and POKE31,3Ø.

If we're on page 2 we get our hi byte at $DE and $DF, which is POKEd in line 25 with $1DØØ. The lo bytes for either page are the same and are found in $1EØØ, which addresses are given to us from $CE and $CF --- the POKES for this were also done in line 25.

The reason for POKE36893,32 and POKE36894,3 in line 25 are to replace $9Ø1D and $9Ø1E in TEST E (CALL36934) with $32Ø so line $9Ø1C changes from $9Ø1C JSR $F411 (HPOSN routine) to JSR $Ø32Ø where the YTABLE accessing routine is located after being POKEd in, in line 25. This is the only change in that animation program necessary to convert from HPOSN usage to YTABLE usage.

To clarify table usage, notice that the 3 different sections of YTABLE are 24 lines of 8 bytes each or 192 bytes long --- 3/4 of a page. It makes sense then that we can index, like with LDA ($DE), Y, to a displacement of 191 past the first byte in that table section. The Y can be Ø-191 (dec.). We will get the appropriate vert. coord's. screen address for the byte Ø column with looking up both the lo and hi address bytes, putting them respectively into $26 and $27, and then referencing screen locations on this line with LDA ($26), Y in the animation program. This is post-indexed indirect addressing. It means the indirection comes before the indexing and the indexing comes after the indirection. We get the address (16 bit) from $26 and $27 and go there and then add a displacement to the address, so if the $26 and $27 bytes point to $2ØØØ and Y is $9, we will be loading into the accumulator the byte of data in address $2ØØ9.

Now look at the reference manual's hi-res map on page 21. $2Ø8Ø is the base address of the 9th line down at coordinate (vert.) of 8 (don't forget the first line is line Ø). The 9th byte in the page 1 hi byte section of YTABLE ($1ECØ-$1F7F) is $2Ø and the 9th lo byte address byte ($1EØØ-$1EBF) is $8Ø. That's all the "base calculation" it took, once we knew our Y register **indexer** was going to be 8. Remember a Ø indexer displacement will give us the first table byte and an 8 indexer displacement will give us the 9th table byte --- the one we want; so you can see how a Y coord. (vert.) of 8 will give us both the 9th line down and the 9th table addr. byte.

It's a relatively simple concept. If you don't believe it do a monitor list (L) of HPOSN at $F411 and continuing for many screens of garble. I'd rather deal with YTABLE than decipher **those** "runes", right? (Actually, one needn't **understand** HPOSN to use it --- just JSR $F411 and don't sweat it.)

Look through the base addresses at the left edge of the hi-res map in the reference manual. Now browse through YTABLE. Do you see that YTABLE has merely table-ized all this address data? It saves time --- but not space --- unless you kick DOS out of your disk and have extra room anyway.

One thing for sure --- if you ever access HPOSN **a lot** in a program requiring speed, YTABLE is the best way to go --- because when you boil it all down, one fact stands out: the HPOSN routine is long, cumbersome, and **slow** compared to table look-up.

BLOAD ENCLOSED SHAPES 32

There is an easy way to BSAVE any of 15 vector shape tables onto your own disk. You'll also be given BSAVE address and length info, which you won't need at the time, but jot it down for future reference.

Always keep in mind that after a BLOAD you can find length with ?PEEK(43616)+PEEK(43617)*256 and address with ?PEEK(43634)+PEEK(43635)*256 and your hex addresses can be translated into dec by:

```
(type) CALL-151
(type) *75:(lo) (hi) NED20G (Return)
```

And your dec can turn into hex with use of:

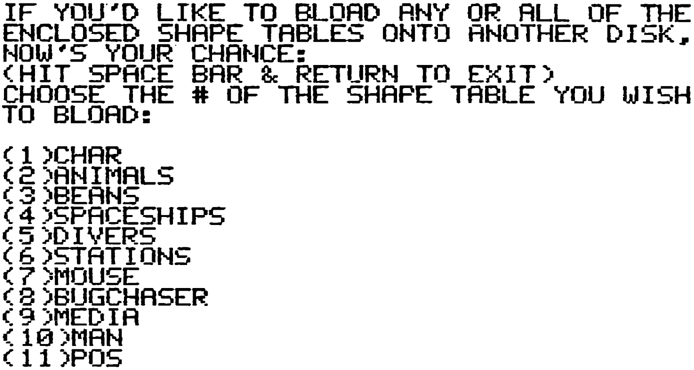
```
(if ?PEEK (104) gives 8)
NEW
(type) (dec#)A      (Return)
(type) CALL-151      (Return)
(type) *803.804      (Return)
```

If ?PEEK(104) is 64 make your last line *4003.4004 and if ?PEEK(104) is 96 make your last line *6003.6004.

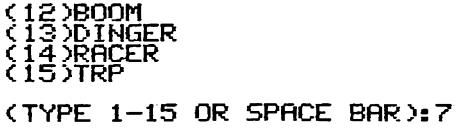
The BLOAD Enclosed Shapes program can be RUN manually or simply gotten to by menu; it's 1 of 28D.

Here's what you'll see:

PRINTOUT #65



```
IF YOU'D LIKE TO BLOAD ANY OR ALL OF THE
ENCLOSED SHAPE TABLES ONTO ANOTHER DISK,
NOW'S YOUR CHANCE:
(HIT SPACE BAR & RETURN TO EXIT)
CHOOSE THE # OF THE SHAPE TABLE YOU WISH
TO BLOAD:

( 1 )CHAR
( 2 )ANIMALS
( 3 )BEANS
( 4 )SPACESHIPS
( 5 )DIVERS
( 6 )STATIONS
( 7 )MOUSE
( 8 )BUGCHASER
( 9 )MEDIA
( 10 )MAN
( 11 )POS
```



```
(12)BOOM
(13)DINGER
(14)RACER
(15)TRP

(TYPE 1-15 OR SPACE BAR):7
```

You'll get disk-switch commands and length and address data. Use these shapes to create scenes with the other programs on 28D, like option 9 in 4 of 28D or similar routines in 8 of 28D. Have fun!

ASSEMBLERS, ASSEMBLY, LISA 33

I tend to find LISA extremely useful and fun to use, and I'm used to "her", so I haven't "played the field" too much. If you want revues on which assemblers are best for what applications, I suggest you consult Peelings, the best software review magazine around.

My LISA gets ornery if I try to make printer print-outs of source codes that are less than full-file, so I have to resort to screen print-outs rather than LISA-controlled listings. But a full source-code list-out usually comes off okay. My LISA doesn't always boot unless another disk gets booted first. Neither of these idiosyncracies have led to any misery on my part --- the cures are infinitely simple.

The usual problem with assemblers is either operator error or documentation ambiguities.

The one big error that most of us make a lot, at least at first, is assembling our code in places that clash with our assembler's own "personal" addresses. The result is bombed assemblers that must be rebooted, and programs that must be redone. (Watch your language!)

For instance, FILL (source for FILL1) starts with ORG $9000 and OBJ $800. If I leave out the OBJ $800 and start assembling, I'll soon bomb LISA (once I try "A" to **assemble,** it's all over) if my program isn't short. I'm supposed to assemble between $800 and $1800. OBJ $800 tells LISA to assemble at the normal $800 address but ORG $9000 tells her that I'll be **using** the program at $9000. So once I'm done writing the program, what do I do?

1) Assemble? No, WRITE FILL onto a separate disk. This will save the source code.

2) Now A for Assemble and cure any errors that manifest.

3) Now BRK once or twice, as LISA commands, and list (monitor L) to find the length of the binary file created -- jot it down.

4) Now *7003G to warm start LISA without messing up any code.

5) Now CTRL D BSAVE FILL1, A$800, L$320, S6 and you've saved your binary file (separate disk!)

6) Now CTRL D PR#1 for printer turn on.

7) Now L to list the program on the printer.

8) Now boot your binary file disk and BLOAD FILL1, **$A9000** and BSAVE FILL1, A$9000, L$320 to get the program where I want to run it. It was at $800 before. What good did ORG $9000 do when I was doing the assembly? It made the operand codes (the addresses upon which to perform the opcode instructions) be relative to $9000, not to $800. This means the program won't work at $800 (unless it's portable) and will work at $9000. A line that was JMP LOOP in source code would have been JMP $822 or something in a binary file

disassembly if ORG $9000 was omitted. This is absolute, not relative addressing and is non-portable, so the program wouldn't have run at $9000 until you went in and changed these types of lines. With the ORG $9000 in there the JMP LOOP would have disassembled (from binary file) as JMP $9022.

9) To fix errors in a source file and its binary "accomplice", CTRL D EXEC FILL, S6 and go ahead. Then re-SAVE it and the binary file result.

There are a couple of tricks I use at times: I put 2 BRKs before the compulsory END line so that when I search with monitor L commands for the end of the assembled object code, I find it fast --- there are 2 double zeros as markers. Also, I don't put comments right into the source-files, I **write** them on the printer print-outs in ways that are much easier to see what the program is all about than space-eating source-code comments which are much harder to adjust or change than penciled words.

I generally write the source codes out on paper before assembling if the programs are complex. I run mentally through what is supposed to be happening as each command is encountered. If everything clicks I start up LISA.

When there are similar subroutines I try to make one dual-purpose subroutine suffice. If it won't, I usually don't need to retype the routine. I list the similar routine and go through it with edit commands after first hitting I (INSERT) and the right line number.

I use file names that work well for me and are useful to others. To make the file EXPLOSION (CALL5472) make its noises, all one does is BRUN the file or BLOAD it and CALL5472.

I make source name and binary file name unmistakably similar, like FILL, FILL1, or maybe TEST32, FAST ∧ 64-LN SCROLL (TEST#32).

The reason one assembles is that it's the easiest and fastest way to create good mach. lang. programs.

65Ø2 INSTRUCTION SET

34

(NOTE: Commands I haven't found relevant for graphics or sounds have been omitted.)

ADC Before you add something in mach. lang. clear the carry first (CLC). This instruction adds the data + the accumulator + the carry and stores the result in the accumulator. It may be used in either decimal (SED) or binary mode.

AND I use it to Ø bits, usually. AND #$7F zeros hit bit. The output bits are 1 only if both input bits are 1.

ASL Shift all 8 bits in a byte left 1, which means hi bit drops off (and lands in the carry). The Ø bit gets a Ø put into it. Screen bytes are backwards so an on-screen shift left will be right on the CRT.

BCC Branch if carry clear; detects accumulator less than the data after CMP, CPX, CPY.

BCS Branch if carry set; detects accumulator greater than or equal to data after CMP, CPX, CPY.

BEQ Branch if equal to Ø means "if Z (zero) flag is set." Use to detect accumulator=data, after CMP, CPX, CPY.

BIT Performs AND between accumulator and memory conditioning overflow flag, negative (sign) bit, and Z (Ø) bit. Result not stored. BIT $CØ61 reads button #Ø, and BPL afterwards finds out if the button was pushed by seeing if bit 7 was Ø (7 automatically stores in N, which is tested by BPL).

BPL Branch if negativity flag (N) is Ø. See above.

CLC Clear carry, before ADC or sometimes before shift is done to test bits.

CMP Subtract data from accumulator but don't store results. Condition carry, neg. flag (sign), and zero flag. Use before BCC, BCS, BEQ, BNE, etc. Find out equal to, less than, or equal to or greater than.

CPX Like CMP but it's data and X register, not data and accumulator.

CPY Like CMP only it's data and Y register, not data and accumulator.

DEC Decrement memory by 1.

DEX Decrement X by 1. Great for loops and counters.

DEY Decrement Y by 1. Great for loops and counters.

EOR Exclusive-OR; I use it to XDRAW or to complement screen bytes (EOR #$7F). Result stored in accumulator. Output bit is 1 only if input bits are **different.**

INC Increase memory by 1.

INX Increase X by 1.

INY Increase Y by 1.

JMP Jump to specified address.

JSR Jump to subroutine (mach. lang. version of GOSUB). An RTS is like a RETURN and will bring you back.

LDA Load accumulator with data.

LDX Load X register with data.

LDY Load Y register with data.

LSR Shift all 8 bits in a byte left (on video this will appear right ---the bytes are backwards). The one bit drops into carry, the 7 bit gets a 0 put into it.

NOP No operation, allows you to dump commands without re-doing a file; just put $EA in place of byte.

ORA Logical OR; I use it mostly to set bits. Output bit 0 only if both input bits 0. ORA #$80 sets hi bit.

PLA Pop top word of stack into accumulator (stack point is incremented), can be part of a POP (in BASIC terms) instruction, or can store data in the stack ($100-$1FF).

PLP Pop top word of stack into status register and stack pointer is incremented. Status register has all the flags like N, V, B, D, I, Z and C (carry).

ROL Rotate left one bit, 7 goes in carry and carry goes in bit 0. Use for shift-animation.

ROR Rotate right one bit, 0 goes in carry and carry goes in 7 bit. Use for shift-animation.

RTS Return from subroutine, like BASIC's RETURN.

SBC Subtract with carry, so SEC (set carry) first. The result is left in accumulator. Data subtracted from accumulator, with carry. Binary or decimal mode.

SEC Set carry; use before SBC.

STA Store accumulator in memory. Accumulator unchanged.

STX Store X in memory; X unchanged.

STY Store Y in memory; Y unchanged.

TAX Transfer accumulator into X.

TAY Transfer accumulator into Y.

TXA Transfer X into accumulator.

TYA Transfer Y into accumulator.

Here are some memory addressing modes (for LDA):

LDA addr.	zero page direct
LDA addr.,X	zero page indexed
LDA (addr.,X)	pre-indexed indirect
*LDA (addr.+1,add.),Y	post-indexed indirect
LDA addr. 16	absolute
LDA addr. 16, X or Y	absolute indexed

*(Whatever zero page address ($26) is between parenthesis is lo byte, and the next zero page address after it is hi byte --- of the indirect address being referred to; **the above form is hi, low.**)(The addr.+1, addr. and is standard way of showing 16 bit address.)(An example would be if $26 was 00 and $27 contained $9, then LDA ($26),Y would mean load accumulator from $900 plus whatever Y was. If Y register contained $8 then the final address here would be $908.)

POKES, PEEKS AND CALLS FOR EVERY PURPOSE 35

CALLS (and JSRs and JMPs)

1) 62923 ($F5CB), then PEEK into 224, 225, and 226. This is HFIND and it's used after DRAWing to find out where you've been left (hor. lo, hor. hi, vert. found with PEEKS)

2) 63Ø69 ($F65D) XDRAW

3) 62977 ($F6Ø1) DRAW

4) JSR $F3D8   HGR2

5) JRS $F3E2   HGR

6) 62454 Clears page (in $E6) to last hplotted color ($F3F4)

7) 6245Ø Clears page (the one specified in $E6 page 1 is $2Ø, page 2 is $4Ø) to black. ($F3FØ)

8) —868 Clears current line from cursor to right margin.

9) 62481 ($F411) HPOSN

1Ø) 62551 ($F457) HPLOT

11) 62554 ($F45A) PLOT

12) 62778 ($F53A) HLINE

13) JSR $F465 INTX increments /decrements hor. screen coord. by 1; N-flag decides which

14) JSR $F467 DECRX decr. hor. coord. by 1.

15) JSR $F48A INCRX incr. hor. coord. by 1.

16) JSR $F4D3 INTY incr./decr. hor. coord. by 1 according to N-flag setting. BPL is branch instr.

17) JSR $F4D5 DECRY decr. vert. coord. by 1.

18) JSR $F5Ø4 INCRY incr. vert. coord. by 1.

19) JSR $F5CB IPOSN makes external cursor at $EØ, $E1, and $E2 equal to internal cursor data at $26, $27, $3Ø, $E5

2Ø) JSR $FBDD BELL1 beep speaker.

21) JSR $FB1E PREAD reads game controller (pass which one, Ø—3, in X register); Y register gets the result.

22) JMP $3DØG use scan RTS when resetting after a mach. lang. prog. that RTS doesn't return you from

23) 54915 Clears stacks so no FOR-NEXT end RETURN errors will accumulate.

24) 65Ø68 Monitor memory move CALL. Zero Y and store start lo/hi in 6Ø/61 (dec.), and ho/hi in 62/63 and destination low/hi in 66 (dec) and 67 (dec).

25) 1ØØ2 Re-enable DOS commands

PEEKS (#s in ( ) get PEEKed at)

1) (—16384) Byte > 127 if keyboard hit (clear this with POKE-16368,Ø).

2) (38)+(39)*256 Latest hi-res base address ($26 and $27)

3) (−16336) Clicks speaker.

4) (−16287) Reads PDL button 0, is > 127 if pushed.

5) (−16286) Reads PDL button 1, is > 127 if pushed.

6) (104) Hi byte of start of program pointer, is 0 for normal or no graphics, is often 64 or 96 for starting program at $4000 or $6000. (103) would be 1.

7) (222) Error code (0-255).

8) (218)+(219)*256 Line # of error.

9) (230) ($E5) Draw on page one is 32 ($20), draw on page two is 64 ($40).

10) (232)+(233)*256 Address of shape table.

11) (43634)+(43635)*256 Address of BLOADed file

12) (43616)+(43617)*256 Length of BLOADed file

13) (228) Current color byte (mask). ($E4)

14) (224) Current hor. lo. ($E0)

15) (225) Current hor. hi. ($E1)

16) (226) Current vert. coord. ($E2)

17) (249) Rotation ($F9)

18) (229) ($E5) Integer part of hor. screen byte.

19) (48) ($30) Bit position table byte.

20) (28) ($1C) Color masking byte shifted for odd address

21) (214) ($D6) All commands=RUN if > 127; normal is < 128.

22) (231) ($E7) Scale.

23) (234) ($EA) Collision counter (change means collision).

POKES

1) 214,214 All commands=RUN (2nd # must merely be > 127).

2) 230,32 Draw on page one.

3) 230,64 Draw on page two.

4) 1010,10
1011,0 } Reset = Reboot
1012,0

5) −16368,0 Reset keyboard strobe so we can read keyboard again.

6) 33,33 Display (video monitor's) window width to set before editting --- you'll see why when you use use ESC keys and arrows to go through BASIC lines. (33, 40 is normal setting; Reset will get 33, 40)

7) 33,30 Window setting during Capture in DOS manual.

8) −16304,0 Display graphics.
−16303,0 Display text.
−16302,0 Display full-screen text or graphics.
−16301,0 Mix text and graphics.
−16300,0 Display page 1
−16299,0 Display page 2
−16298,0 Display text of same page as graphics were.
−16297,0 Display graphics of same page as text was.

9) 216,0 Reset error flag so normal error messages will happen.

```
10) 1010,152
    1011,216   } Reset=Continue
    1012,125
11) 1010,242
    1011,3     } Reset=Freeze
    1012,166
```

36 USE OF LISA-EXECABLE SOURCE-CODE TEXT FILES

This system contains lots of source files for scrolling, white line fixing, filling with color, and especially for a whole collection of different hi-res drawing and animation routines for hplot-shapes, block-shapes, and vector shapes. The source code for **scanning** sections of screen to be defined as block shapes is also included. **Listings** of a sample sound source-code called MULTIPLE LASER! are included as well.

The way to use these is to boot LISA or any other compatible assembler and once you see the exclamation point that says LISA's ready, type CTRL D EXEC (file name of source-code test file)(Return)

Now hit L to list, M to modify, D to delete, I to insert, A to assemble, WRITE (file name) to re-WRITE, CTRL D BSAVE (file name slightly changed), A$addr., L$length (Return) to save binary file, CTRL D PR#1 (Return) and L to print-out listing, BRK to get into monitor, *7003G for warm start and *7000G for program-erasing cold-start.

Study the source-codes using the line-by-line explanations given in this manual. These files don't teach you to suddenly make Raster Blaster --- what they do is teach you most of the basics you need to know in order to take it from there on your own. You know how to create various shape types, various animation types, various coloring schemes, scrolling schemes, YTABLE usage schemes, sequence sounds & noise & violin & music schemes. These programs are a springboard to use so that you may go unaided in any graphics direction you desire --- begin getting good results soon with no more fumbling or frustration.

FOR THE INEXPERIENCED 37

(For People Who Don't Hardly Know Nothing)

I still remember when I knew very little about all this --- I remember asking people, stores, Apple Hot Line, etc. questions which they often couldn't answer. I remember searching in vain through Apple-centered magazines and often finding out very little.

I remember old Don Fudge being Howard Peale and saying "I'm mad as hell and I'm not going to take it any more", about the sorry state of information dispersal in the entire microcomputer industry. It's not only Apple.

But, in spite of all, I'm still in love with the Apple Computer and my feeling is that I want this computer to really make it and to also be the one microcomputer that remains a permanently viable fixture of the small computer industry.

I'd like, therefore, to take a more right-brain-wholistic view of the whole Apple scene. It's not that unlikely that IBM and some of the other competitors are all ready to take great strides in micro development and hire 1000 programmers to back up their hardware with software. Rumor has it that IBM's already planning to unveil 3 small Apple competitors this year or next year.(Another rumor has it that the small IBM micros will run Apple software, which would be great for Apple software producers, but bad news for Apple!) And anyone who'se been watching the micro markets has noticed that a whole swarm of competing viable micros are in the process of converging upon the industry from both Japan and local hardware companies, so we can't afford to goof all this up.

In an immediate-profit context, it makes great sense for all the software manufacturers and programmers to jealously guard each and every piece of information they come up with, and lock all new programs, utilities, ideas and POKES and CALLS in an H-Bomb proof safe. It's good business to hide in the closet with information, asking large sums of money for each tiny bit of data you're so kind as to dole out. Profit is what business is all about, and I have no problem with that.

The problem comes when these short-range profit motives are not tempered with a more wholistic, realistic view of the situation. In other words, along with good business practices there needs to also be good **foresight.** Scrambling for bucks is a very poor long-range strategy, if Apple is to remain viable.

I suggest instead a sharing, cooperative attitude that causes Apple software to develop and improve too fast for any megacorporations to swamp.

People who have written helpful articles in the software magazines (*Apple Orchard, Call Apple, Softalk, Micro, Nibble, Applesauce*) are to be commended. This manual is my contribution to Apple information.

The reason I wanted "inexperienced" people to read the above is that I believe that the **context** in which programmers hold all this "micro-mania" is quite important. Capitalism is a fair economic system, but buck-obsession is another matter altogether. Each and every graphics programmer so far has had to re-invent the wheel and start almost from scratch. This is not a healthy situation --- no matter who says what. Our society and national progress are based on sharing and cooperation **as well as** competition. That's the angle that seems to be most often forgotten in the scramble for bucks --- a symptom of an economy in trouble.

In the short run one of the experts doing the fanciest shoot-em-up yet will make a few extra bucks with keeping everything he does secret. On the other hand, in the **long** run the publication of not only applications software (like 3-D graphics packages) but programming method information and other data (like new programming discoveries) will not only benefit himself and other programmers, but the Apple software industry, and ultimately the viability of Apple as a computer is increased. **And isn't this the long-range goal we all share?**

I admit this is an easy consideration to lose track of in the hustle and bustle of an average workday, but I can't help but feel that if too many of us "forget" too often, we'll find ourselves someday in the uncomfortable position of being on the outside looking in.

Enough said.

If you know little about programming and wish to make extensive use of this manual, do yourself a favor and study BASIC for a bit until you feel comfortable with it --- it's easy and doesn't take long. Then try to use one of the assembly books to give you a basic idea of what 65Ø2 instructions are all about --- look at the sample programs and figure out what's going on. And read Chapter 34 of this manual. By doing the above you'll be in the best position to profit from the contents of this manual.

This manual was written for the amateur programmer, or the open-minded beginner who's willing to study the Apple manuals and familiarize him/herself with BASIC before proceeding, or the professional who knows all about BASIC, other languages, DOS, hardware, circuitry, and whatever else, but just never got around to graphics before.

This manual was written in a way that hopefully doesn't assume too much knowledge on the part of the reader. By far the biggest single problem of the **somewhat** useful, somewhat informative programs or mach. lang. routines in most micro magazines is the tendency of authors to act as though their audience were at the same level of knowledge as they. This is seldom the case. People reading those articles are usually **attempting** to attain such knowledge, but do not already possess it.

I tried to write in a way that takes not too much for granted. When I hear in my head "why tell them those last 6 lines are a delay loop --- any dork can see that --- it's a waste of time!", I just remember back to when I didn't know a delay loop from a tube of Preparation H. So more experienced people will find more explanations than they need, while amateur graphics people may find that my soup is seasoned about right. A rank beginner, no matter **what** manuals they read, will feel in the dark until BASIC and a couple of rounded teaspoon-fulls of assembly are swallowed and digested.

So what should you do if you "don't hardly know nothing" and you don't know BASIC or anything and you **still** want to make use of this package? Well, if you're open to learning, then go through the manual reading, running programs, drawing shapes and scenes and coloring them with Palette, run sounds files and Superfont and scrolling programs and Instant Graphics (Block Shapes), play the sample game a lot, and list out programs and see if you can figure anything out once in a while. Perhaps you'll get intrigued with some aspect of all this and go back and learn BASIC someday and continue using this package, but now more as the learning tool it was meant to be. If not, that's okay, there are plenty of interesting things to keep you busy playing, drawing, creating.

But just how much of an "applications" package is this? Is it as useful as it is informative? Yes, very much so.

· Not only do you get the tools for creating shapes, sounds, color-filled pictures, etc., you also get the tools for creating hplot-shape sequences, block-shape sequences, shift-animation and 2-page flipping sequences, font styles (if you know BASIC) and font creations, and shapes of all types and colors and sizes.

But I'll give an example: In the November 1979 MICRO, Bob Bishop was kind enough to give people a hi-res compression program that allowed full-screen pictures to be compressed so that they'd store in a fraction of the space when on disk or tape. A 34-sector binary file was now only 1Ø sectors or less, many times. The idea was to get more pictures stored on a disk.

Since then Bishop and others have helped people learn about hi-res colors and the result has been an abandonment of the limitation of 6 hi-res colors. My Palette program has hundreds of possible color patterns and Edu-Paint claims billions of colors.

But the evolution of shape drawing and handling methods and the advent of color-fill algorithms has changed things. Bishop got 33 pictures on his famous Super Slide Show. But recent adventure games are getting 1ØØ or more pictures per disk. Compression methods may be at the heart of some of these multi-scene adventure games, I couldn't say. I neither buy many adventures nor ever attempt to "bust" protected programs --- I find the idea very distasteful --- creation is about 1,ØØØ.ØØØ times as much fun as sneaking into someone else's programs and having illicit looks around. Try each yourself --- and then see if I'm right.

But "compression" doesn't seem to be indicated in the newer methodologies. Combinations of hplot, block and vector shapes would probably be the most dynamic as far as quality of picture produced, and hplot shapes would be best if big shapes with lots of straight lines are involved. In any case, fill routines such as the one in this system (FILL1, from Palette) are used to color an entire scene once the scene is drawn. At 6 (or even 2!....see A of 28D) bytes per "seeding" it wouldn't take much memory to color an entire screen). (One could even limit most seedings to 2 bytes, X and Y coords., if the same colors were used a lot --- merely a double FOR-NEXT needed here.)

One of the most economical things about scenes built of shapes is that you can create an enormous variety of scenes with a few dozen shapes. You needn't turn the shapes into permanently stored scenes --- all you need to do is just draw the shapes in various positions (and for vector shapes, various scales and rotations) on the screen. A shape table 3ØØØ bytes long would go a long way here. Hplot shapes for long straight room lines or perspective wall and corner lines, and block or vector shapes for more intricate figures would fill the bill nicely, without filling the memory or disk too quickly.

Two to six-byte fill-seeding would finish the job and the result would be a heck of a lot of possible colorful scenes on one disk. From 1Ø sector B&W in 1979 to hundreds of scenes full of intricate shapes and wild colors in 1981, that's the way Apple software is progressing.

However, if **you** want to be all ready to do this new color-filled shape-scene type of thing, how do you begin? What do you do? Who do you ask? What you do read?

Perhaps you can now see the purpose of this package. The short-range purpose is to give all would-be graphics programmers the information that they've been searching and wishing for so that they needn't all waste time re-inventing the wheel.

But the second one is one based upon the foresight context, mentioned earlier. More than any other reason, this package was created to help make the Apple microcomputer a permanent viable entity. How could I be in love with this computer and do any less?

SUMMARY 38

It is my hope that this system will contain adequate knowledge, tips, programs, examples, routines, explanations, ,source-codes, applications programs, food for thought, and so on; adequate to allow the serious graphics studier to be able to use the package to do whatever it is s/he desires in the graphics arena. The only subjects I find noticeably absent are 3-D graphics (there are plenty of such packages around) and vector shape **drawing.** Apple's Applesoft Manual already explains the subject of vector shapes and I have already developed a package for the drawing and animation of vector shapes called **Super Shape Draw and Animate,** which I recommend for anyone who desires application/information/utility programs in that area. There are vector shape examining, animating, 2-page flipping animation for vector shapes, and other vector programs in this present system, mostly on 28A (3, G, H, I), but not drawing or editing programs. As you read more and more in the manual you'll become increasingly aware of why I concentrate on block-shapes and hplot-shapes so much. If you need more vector shape **drawing** information, either get the above package or read the Applesoft manual about vector shapes. (However, 5 of 28C will convert **anything** to a vector shape with great ease!)

If you'd like more sound programs, or source codes for the sound or scrolling programs in this system, we have an "Action Sounds and Hi-Res Scrolling" disk available. If you liked where I was coming from with the "samples game", we have dozens more games with that type of slant, most of which are hi-res and action-filled, or at least strategy filled, which is often more enjoyable. Avant-Garde Creations has business, educational, utility, games, informational software available.

People have been telling us what type of software they'd like to see --- we've been absorbing the list of these desires and recommendations for over a year. One of the most-often cited needs was for a decent graphics/informational/applications package that was truly informative and allowed programmers getting into graphics to circumvent re-inventing the wheel, which every graphics programmer **until now** has had to do.

We've tried to provide that which has been solicited in a form that's as convenient, cheap, easy to use and understand as possible. If there are suggestions for improvements or future packages, additions, bugs, etc., let us know --- we're open to suggestions and new ideas at any time.

If you know BASIC, you should now have the tools to create all kinds of fine games/graphics utilities/etc.; and if you know assembly, and can use the many mach. lang. routines in this system in dynamic and strategically and visually stimulating ways, you should be well on your way to the creation of some really wild stuff.

If you're merely artistic and wish only to create colorful scenes (perhaps with added sounds or music --- such as violin sounds), and shapes that you or others will scroll, animate, or use as adventure scenes for hi-res adventures, fine and good. Whatever your graphics application, this can be your foundation.

Have a great time with this system!

—Don Fudge

COMMANDS THE MANUALS NEVER TOLD YOU

39

1) GOFLUB When you know darn well that the routine you're being sent to is **not** going to do what you want it to.

2) DELI You CALL this routine when you run out of refrigerator goodies in the midst of a heavy programming session.

3) EITHER/OR This BASIC COMMAND Is one you can expect from the spouse if you program too much and pay attention to him/her too little --- **either** you wise up **or** s/he splits.

4) VAL: A CALL you make to *CALL APPLE* is likely to end you up with this particular variable.

5) PDL You jump to this routine if you find out who spilled the kool-aid on your Apple.

6) FLASH This routine keeps all the females in the neighborhood on their toes and screaming.

7) WAIT This one is automatic, you needn't CALL it, just sort 1000 names alphabetically and like magic you'll be put into the WAIT mode till the cows come home.

8) CLEARHOME You'll be CALLed on this periodically if you allow enough computer magazines, print-outs, books, and other junk to accumulate.

9) NORMAL This routine you'll probably never see again if you let computer fever take you over as much as many chip-jockeys do.

10) STORE You'll tend to use this routine a lot, but only if it's a computer store --- for any other kind you'll bribe others to go and get the goodies --- you're "right in the middle of a far-out program that's gonna set the world on fire ----".

11) GOHELL After a CALL to the software house that sold you the program that bombed in 3 seconds you'll find yourself giving all sorts of crazy commands.

12) NOTRACE The variable that describes that routine you **know** was in either *Micro* or *Nibble*, but you've looked through every issue 47 times and ---

13) HIMEM This routine is only used when you see your friend Meme.

14) LOMEM Occasionally you substitute this routine for the previous one when you're in such a hurry that first syllables are nearly silent.

15) ONAIRGOTO The way you get around after you successfully completed 12 months of work on a terrifically difficult program --- it works, it works!

16) XDRAW This is the routine you use to sign your big software contract if you're computer literate only, and don't hardly know from the 3 R's.

17) STEP ON The perfect routine for a disk you're mad at.

18) ROT This route is automatic, and will happen to you if you don't get out and get some exercise as well as doing programming.

19) END$ This variable describes your wallet if you can't live without every new adventure that comes out.

2Ø) RECALL This routine is generally what Apple owners feel is the correct one for TRS-8Øs.

21) LOG The variable describes the way chairs feel after the 12th consecutive hour of programming.

22) DIM The way things look after the 413th reassembling of a mach. lang. program fails to make it work right.

23) SIN This variable describes the feeling as you pirate a copy of a program.

24) PLOT If this is what you think that everyone is doing against you, you know you've finally programmed one too many lines --- you've POPped your chips.

25) TAN One thing you'll never get, at your Apple all day.

26) ONERR GONUTS This one runs automatically if you've spent months on error trapping on a program and you sit down with a friend to show him the program; guess what happens when you run it?

27) GOBED The best routine for when you start seeing double.

APPENDIX

SIZE TABLE

NAME	DRAW-CODE	L	M	S	E	X	T	Y	O
		EXACT SIZES (height/width)							
Rectangle (tall)	R*	184/92	144/72	96/48	72/36	48/24	24/12	16/8	12/6
(Wide) Rectangle	W*	92/184	72/144	48/96	36/72	24/48	12/24	8/16	6/12
Square	Q*	92/92	72/72	48/48	36/36	24/24	12/12	8/8	6/6
Triangle	T*	92/92	72/72	48/48	36/36	24/24	12/12	8/8	6/6
Circle	O	184/134	144/144	96/96	72/72	48/48	24/24	16/16	12/12
Ellipse	I	120/240	60/120	30/60	16/30	12/24	8/16	6/12	4/8

*Symmetry Mode only

HEIGHT/WIDTH RELATIONSHIPS OF EACH FIGURE

R	2:1
W	1:2
Q	1:1
T	1:1 (1:2)**
O	1:1
I	1:2

**In *Filled Regressive Mode* T's height/width relationship changes

MAJOR VARIABLE VALUES

A$	=draw what
P$	=what type of line
D$	=what size line or figure
J	=X coordinate of ellipse center
K	=Y coordinate of ellipse or circle center
H	=X coordinate of circle center
D1$	=F if filled with color
X	=PDL(Ø) X-axis coordinate or 'J' insert
Y	=PDL(1) Y-axis coordinate or 'J' insert
S1	=PDL(Ø) button stops 1st 1/3 of wall action, also stops S,E,L,P,R,F action
S2	=PDL(Ø) button, stops 2nd 1/3 of wall action
S3	=PDL(Ø) button, stops 3rd 1/3 of wall action, or signals Symmetry Mode in use
C$	=HCOLORS B,G,V,W,L,O,U,H, default=W,A=abort

INSTANT GRAPHICS (BLOCK SHAPES)

Col.#1
(Draw & Color)

A abort

C color
(optional)
B,G,V,W-
L,O,U,H[1]

B background

D dot ·

G diagonal**

X diagonal**

N diagonal**

Y diagonal**

H horizontal**

U horizontal**

K vertical r.**

V vertical l.**

O circle*

I ellipse*

Q square*

R rectangle*

W rectangle*

T triangle*

P pyrotechnic

S stars/snow**

E enclose**

F frame

L line (goto
column 2)

J block-shape
drawing

M monitor x,y
coordinates

Z end[2]

SPACE Command
Options[3]

ESC Symmetrical
Mode

RETURN Regressive
Mode

Col.#2
(Line)

A abort

G random
placement
of H,V,D,U
(optional)

D diagonal \

U diagonal /

H horizontal

V vertical

P paddles

R random

NOTE:

D,U,H,V
require size,
goto column 3

Col.#3
(Size)

A abort

F fill shape**
(optional)

L large,
(from x,y
to r. edge)

M medium,
(1/2 x,y
to r. edge)

S short,
(1/4 x,y)

E short,
(1/8 x,y)

X shorter yet,
(1/16 x,y)

T even shorter,
(1/32 x,y)

Y very short,
(1/64 x,y)

O shortest,
(1/128 x,y)

*You *must* go to column 3 where you *MAY* enter a fill (F) option, and you *MUST* enter a size.

**Use PDL(Ø) button to stop scrolling or filling.

[1] Colors are: B=black V=violet L=black2 U=blue
G=green W=white1 O=orange H=white2
Press C, then one of the 8 color letters.

[2] After ending the program you may direct your printer to print out your creation.

[3] Choose your MODE:
a) draw: simply type commands.
b) record: hit space bar, type '3' or '11. Then
after you've drawn hit space bar,
type 7 to *save*.
c)play: hit space bar, type '4' or '10'.
d)regressive: if you haven't hit ESC, you're
in it. If you have, type RETURN.
e)symmetry: hit ESC.

MUSIC CODES

KEYBOARD CODES:

```
 (F#)(G#)(A#)    (C#)(D#)     (F#)(G#)(A#)
   1   2   3       5   6        8   9   Ø
ESC  Q   W   E   R   T   Y   U   I   O   P   -
(F) (G) (A) (B) (C) (D) (E) (F) (G) (A) (B) (C)


 (C#) (D#)    (F#)(G#)(A#)    (C#)(D#)
   S   D        G   H   J       L   ;
 Z   X   C   V   B   N   M   ,   .   /
(C) (D) (E) (F) (G) (A) (B) (C) (D) (E)
```

NOTE DURATION CODES

K=𝅝	=240	duration	(whole)
7=𝅗𝅥.	=180	duration	(dotted half)
4=𝅗𝅥	=120	duration	(half)
:=♩.	=90	duration	(dotted quarter)
A=♩	=60	duration	(quarter)
→=♪	=30	duration	(eighth)
←=𝅘𝅥𝅯	=15	duration	(sixteenth)

F=buzz, for sound effect or "cymbals"
SPACE —to record all notes that follow

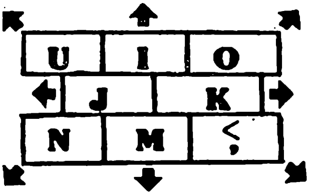
SAMPLE GAME WITH VIOLIN & NOISES

```
                THE MINE FIELDS OF NORMALCY

Q=quit
E=see example, below
P=pickup the symbol you're on
D=drop the symbol you're carrying
move=with game paddles
score: 1000 points for 1 correct line
       3000 points for 2 adjacent correct lines
       8000 points (and game over) for all 3
            correct lines
       -50 points for dropping a symbol on another
           one and losing the 2nd one
       -100 points for hitting a mine
       "chance tornados" happen automatically for
       every 5 mines you step on
```

<	(	♥	E	🔔	♟	1	Σ	)	>
<	(	♥	E	🔔	♟	1	Σ	)	>
<	(	♥	E	🔔	♟	1	Σ	)	>

INSTANT GRAPHICS (BLOCK SHAPES)

```
Block-Shape.
Drawing Commands (J)

Q quit
P plot on/off switch
U northwest
I north
O northeast
K or L east
, southeast
M south
N southwest
J west
```



INDEX

INSTANT GRAPHICS (BLOCK SHAPES)

Col.#1
(Draw & Color)

A abort

C color
(optional)
B,G,V,W-
L,O,U,H[1]

B background

D dot ·

G diagonal**

X diagonal**

N diagonal**

Y diagonal**

H horizontal**

U horizontal**

K vertical r.**

V vertical l.**

O circle*

I ellipse*

Q square*

R rectangle*

W rectangle*

T triangle*

P pyrotechnic

S stars/snow**

E enclose**

F frame

L line (goto
column 2)

J block-shape
drawing

M monitor x,y
coordinates

Z end[2]

SPACE Command
Options[3]

ESC Symmetrical
Mode

RETURN Regressive
Mode

Col.#2
(Line)

A abort

G random
placement
of H,V,D,U
(optional)

D diagonal \

U diagonal /

H horizontal

V vertical

P paddles

R random

<u>NOTE</u>:

D,U,H,V
<u>*require*</u> size,
goto column 3

Col.#3
(Size)

A abort

F fill shape**
(optional)

L large,
(from x,y
to r. edge)

M medium,
(1/2 x,y
to r. edge)

S short,
(1/4 x,y)

E short,
(1/8 x,y)

X shorter yet,
(1/16 x,y)

T even shorter,
(1/32 x,y)

Y very short,
(1/64 x,y)

O shortest,
(1/128 x,y)

*You *must* go to column 3 where you *MAY* enter a fill (F) option, and you *MUST* enter a size.

**Use PDL(Ø) button to stop scrolling or filling.

[1] Colors are: B=black V=violet L=black2 U=blue
G=green W=white1 O=orange H=white2
Press C, then one of the 8 color letters.

[2] After ending the program you may direct your printer to print out your creation.

[3] Choose your MODE:
a) draw: simply type commands.
b) record: hit space bar, type '3' or '11. Then after you've drawn hit space bar, type 7 to *save*.
c)play: hit space bar, type '4' or '10'.
d)regressive: if you haven't hit ESC, you're in it. If you have, type RETURN.
e)symmetry: hit ESC.

SIZE TABLE

NAME	DRAW-CODE	EXACT SIZES (height/width)							
		L	M	S	E	X	T	Y	O
Rectangle (tall)	R*	184/92	144/72	96/48	72/36	48/24	24/12	16/8	12/6
(Wide) Rectangle	W*	92/184	72/144	48/96	36/72	24/48	12/24	8/16	6/12
Square	Q*	92/92	72/72	48/48	36/36	24/24	12/12	8/8	6/6
Triangle	T*	92/92	72/72	48/48	36/36	24/24	12/12	8/8	6/6
Circle	O	184/184	144/144	96/96	72/72	48/48	24/24	16/16	12/12
Ellipse	I	120/240	60/120	30/60	16/30	12/24	8/16	6/12	4/8

*Symmetry Mode only

MAJOR VARIABLE VALUES

A$ =draw what
P$ =what type of line
D$ =what size line or figure
J =X coordinate of ellipse center
K =Y coordinate of ellipse or circle center
H =X coordinate of circle center
D1$=F if filled with color
X =PDL(Ø) X-axis coordinate or 'J' insert
Y =PDL(1) Y-axis coordinate or 'J' insert
S1 =PDL(Ø) button stops 1st 1/3 of wall action, also stops S,E,L,P,R,F action
S2 =PDL(Ø) button, stops 2nd 1/3 of wall action
S3 =PDL(Ø) button, stops 3rd 1/3 of wall action, or signals Symmetry Mode in use
C$ =HCOLORS B,G,V,W,L,O,U,H, default=W,A=abort

HEIGHT/WIDTH RELATIONSHIPS OF EACH FIGURE

R	2:1
W	1:2
Q	1:1
T	1:1 (1:2)**
O	1:1
I	1:2

**In *filled Regressive Mode* T's height/width relationship changes

SAMPLE GAME WITH VIOLIN & NOISES

THE MINE FIELDS OF NORMALCY

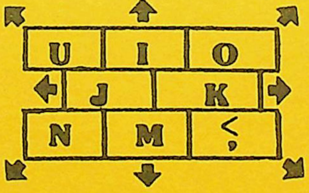
```
Q=quit
E=see example, below
P=pickup the symbol you're on
D=drop the symbol you're carrying
move=with game paddles
score: 1000 points for 1 correct line
       3000 points for 2 adjacent correct lines
       8000 points (and game over) for all 3
            correct lines
       -50 points for dropping a symbol on another
           one and losing the 2nd one
       -100 points for hitting a mine
       "chance tornados" happen automatically for
       every 5 mines you step on
```

<	<	♥	E	🔔	♟	1	Σ	>	>
<	<	♥	E	🔔	♟	1	Σ	>	>
<	<	♥	E	🔔	♟	1	Σ	>	>

INSTANT GRAPHICS (BLOCK SHAPES)

Block-Shape
Drawing Commands (J)

Q quit
P plot on/off switch
U northwest
I north
O northeast
K or L east
, southeast
M south
N southwest
J west



MUSIC CODES

KEYBOARD CODES:

```
 (F#)(G#)(A#)    (C#)(D#)    (F#)(G#)(A#)
   1   2   3       5   6       8   9   Ø
ESC  Q   W   E   R   T   Y   U   I   O   P   -
(F) (G) (A) (B) (C) (D) (E) (F) (G) (A) (B) (C)

 (C#) (D#)    (F#)(G#)(A#)    (C#)(D#)
   S   D        G   H   J       L   ;
 Z   X   C   V   B   N   M   ,   .   /
(C) (D) (E) (F) (G) (A) (B) (C) (D) (E)
```

NOTE DURATION CODES

```
  K=o    =240 duration  (whole)
  7=𝅗𝅥·   =180 duration  (dotted half)
  4=𝅗𝅥    =120 duration  (half)
  :=♩·    =90 duration  (dotted quarter)
  A=♩     =60 duration  (quarter)
  →=♪     =30 duration  (eighth)
  ←=𝅘𝅥𝅯     =15 duration  (sixteenth)
```

F=buzz, for sound effect or "cymbals"
SPACE --to record all notes that follow